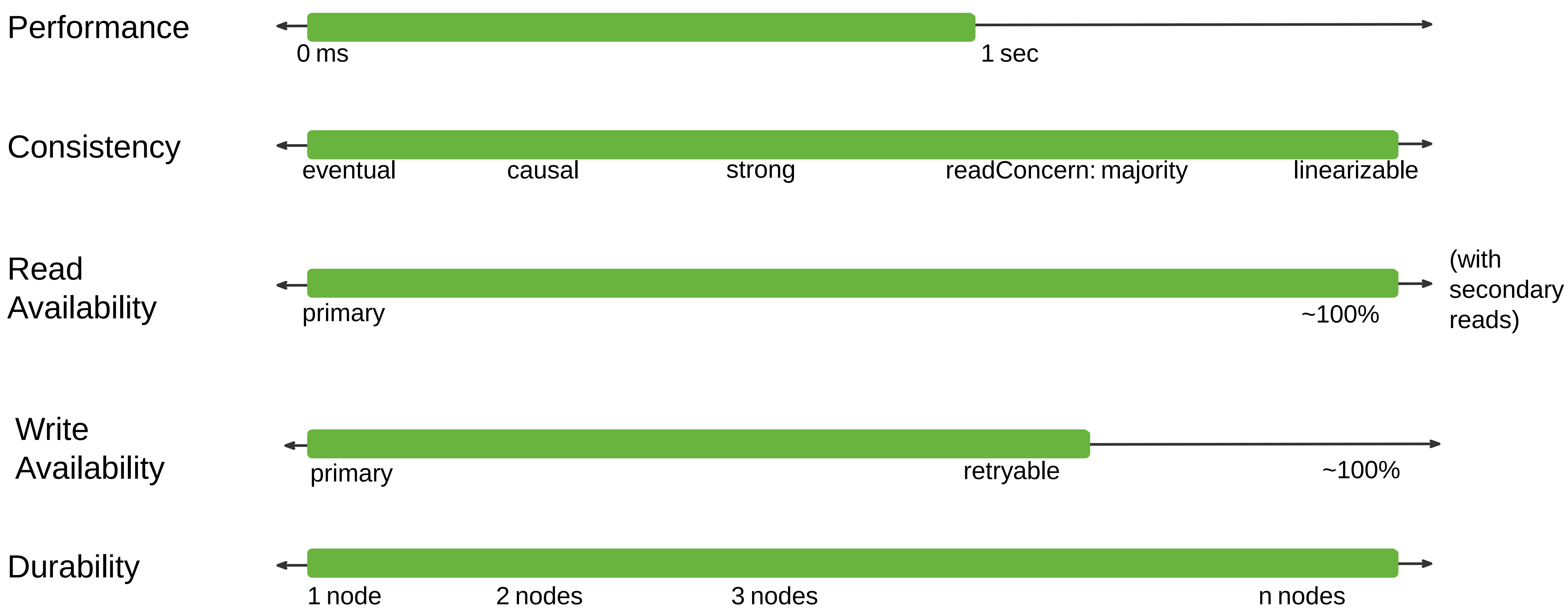
MongoDB Architecture

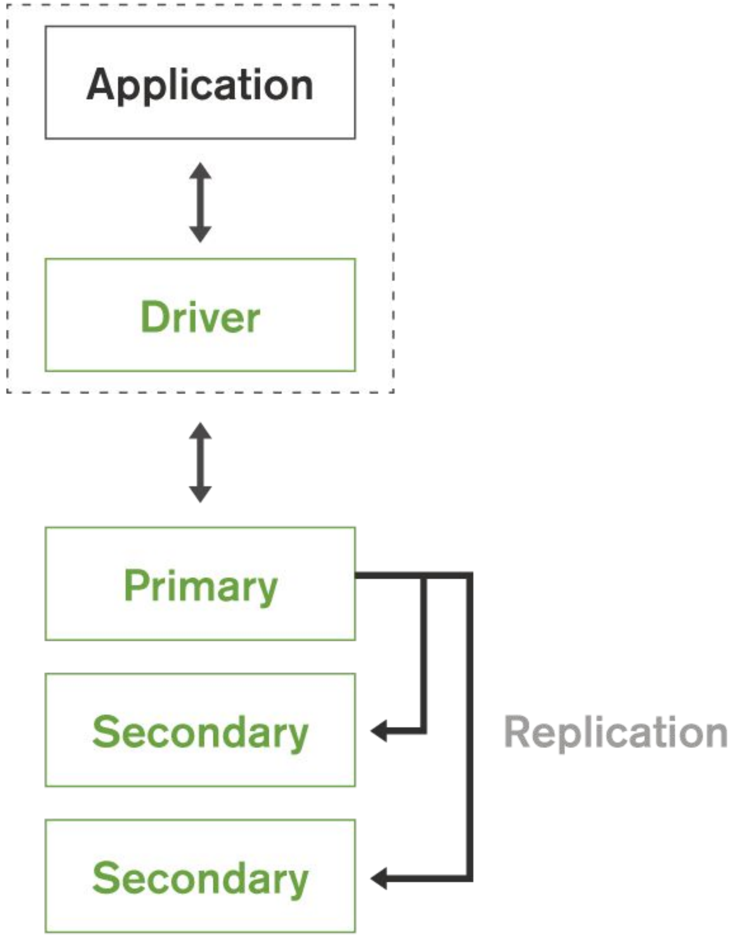
Trade Offs


Performance
0 ms
1 sec
Consistency
eventual
causal
strong
readConcern: majority
linearizable
Read Availability
primary
~100%
(with secondary reads)
Write Availability
primary
retryable
~100%
Durability
1 node
2 nodes
3 nodes
n nodes

Tuning MongoDB to meet Active Active Requirements

Active Active Requirements	MongoDB Capability
Performance	Zone Sharding
Availability	Secondary Reads Retryable Writes
Consistency	Read Preference Read Concern
Durability	Write Concern

Replica Sets



Replica Set – 2 to 50 copies

Self-healing shard

Data Center Aware

Addresses availability considerations:

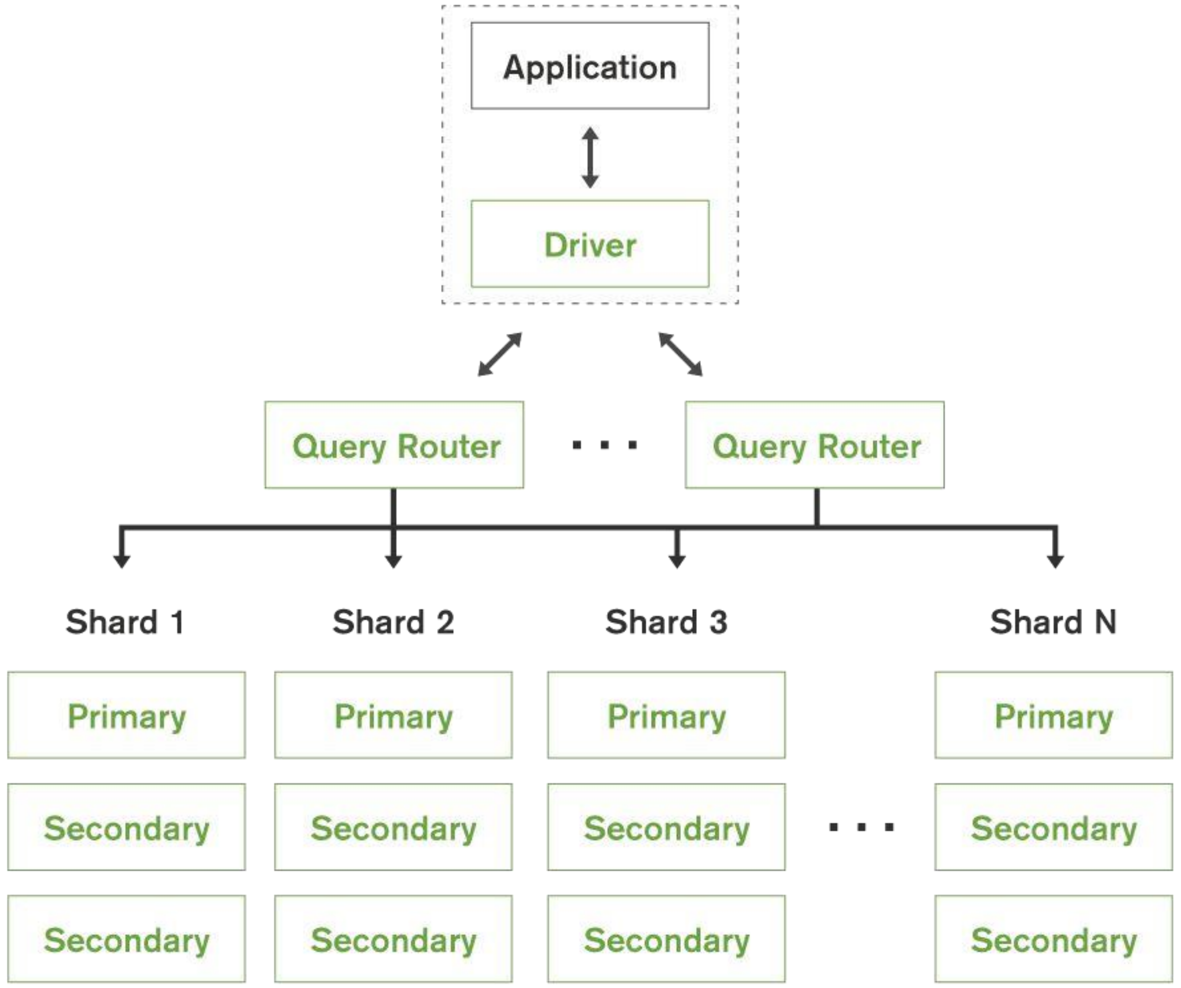
High Availability

Disaster Recovery

Maintenance

Workload Isolation: operational & analytics

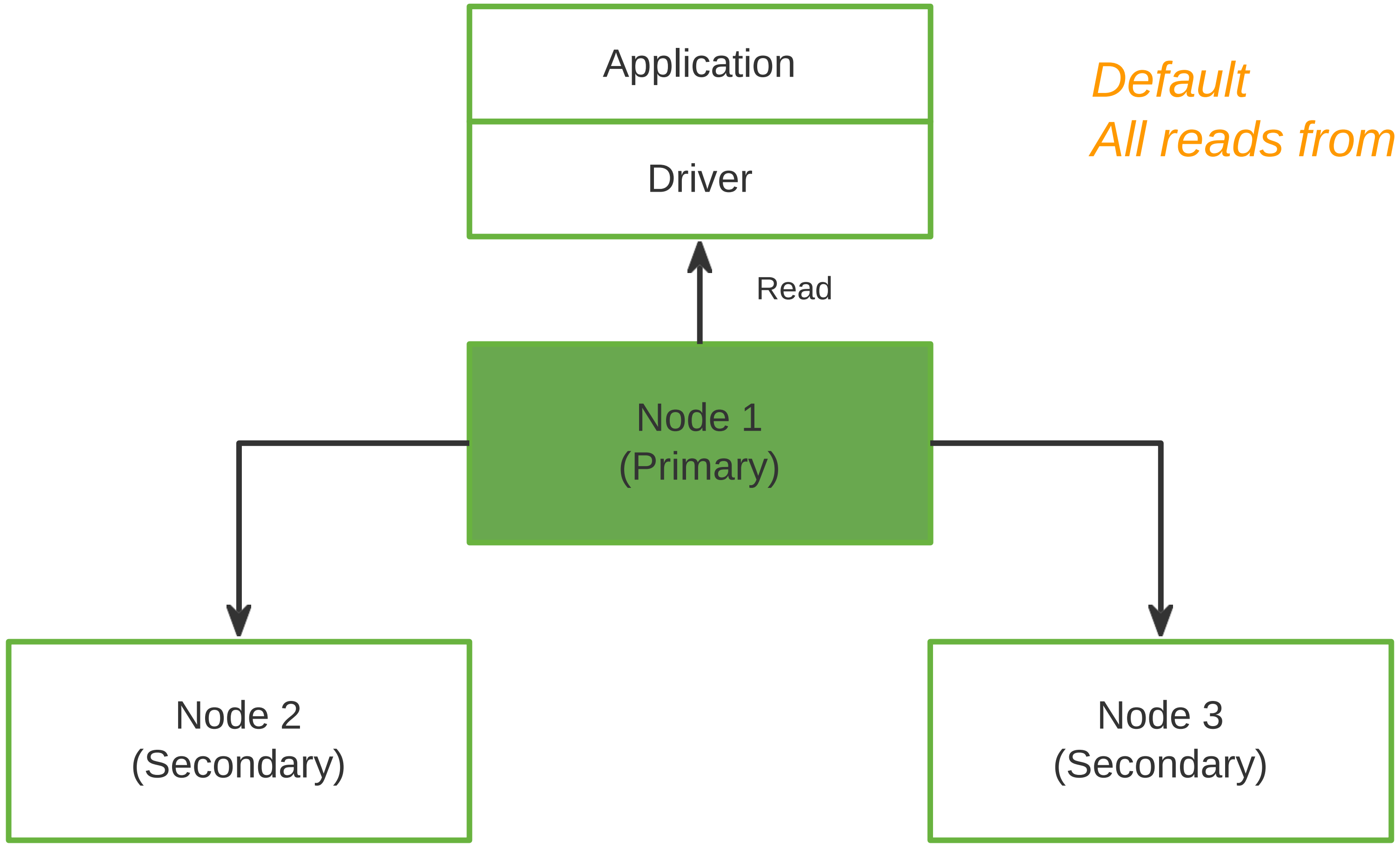
Sharding: Scaling MongoDB



Tuning Read and Write Availability

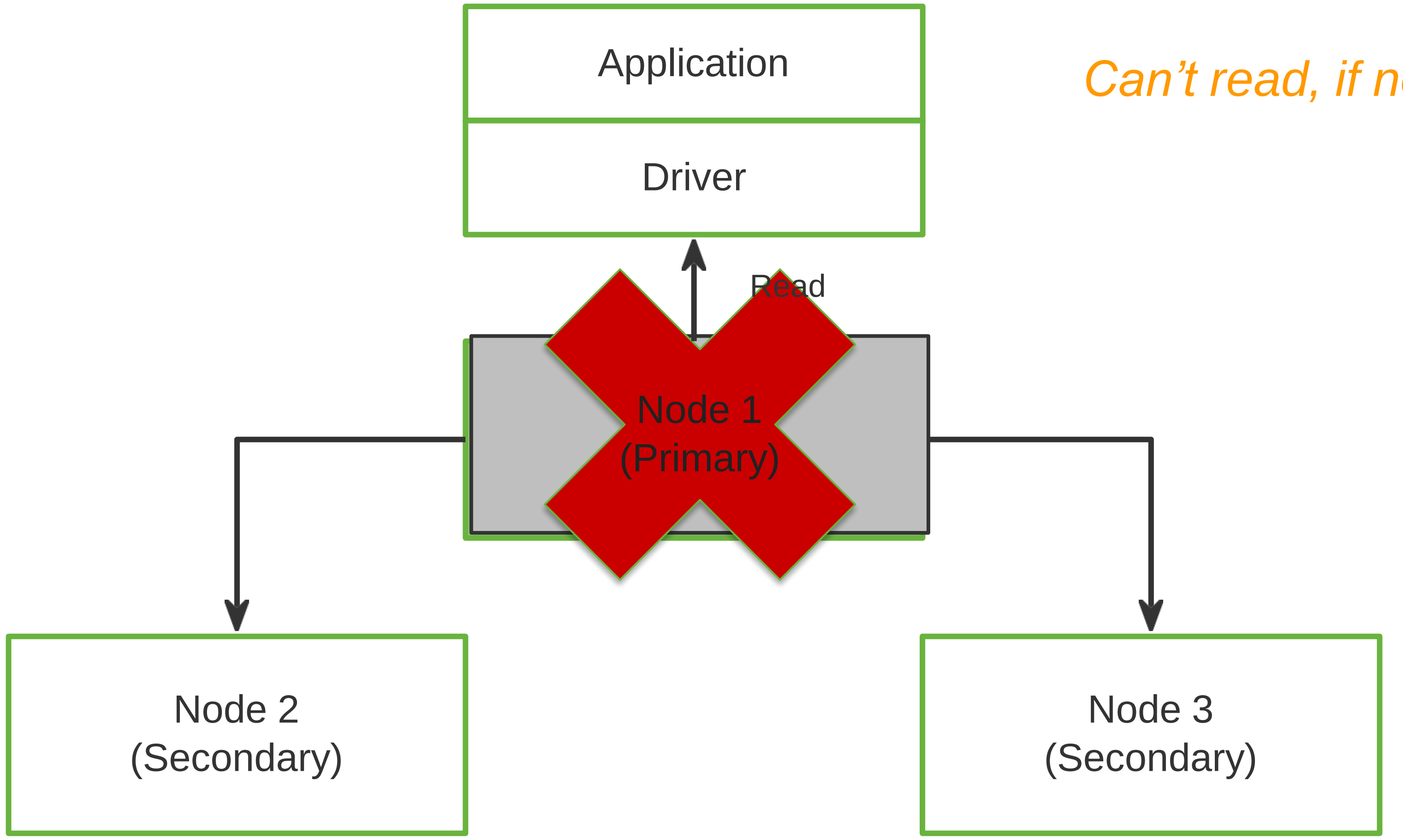
Read Preference
Retryable Writes

Read Preference: Primary



Default
All reads from Primary

Read Preference: Primary



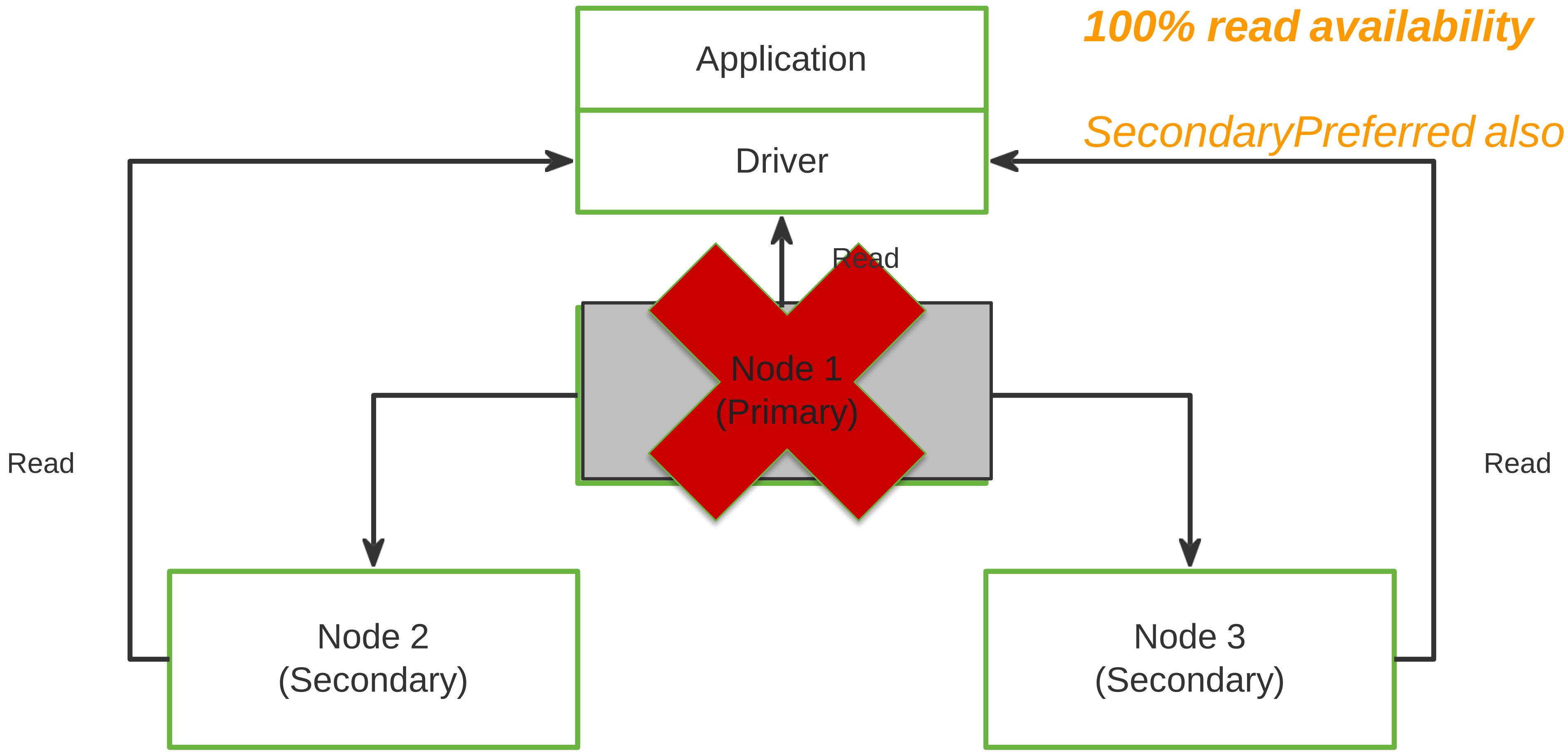
Can’t read, if no primary

Read Preference: PrimaryPreferred

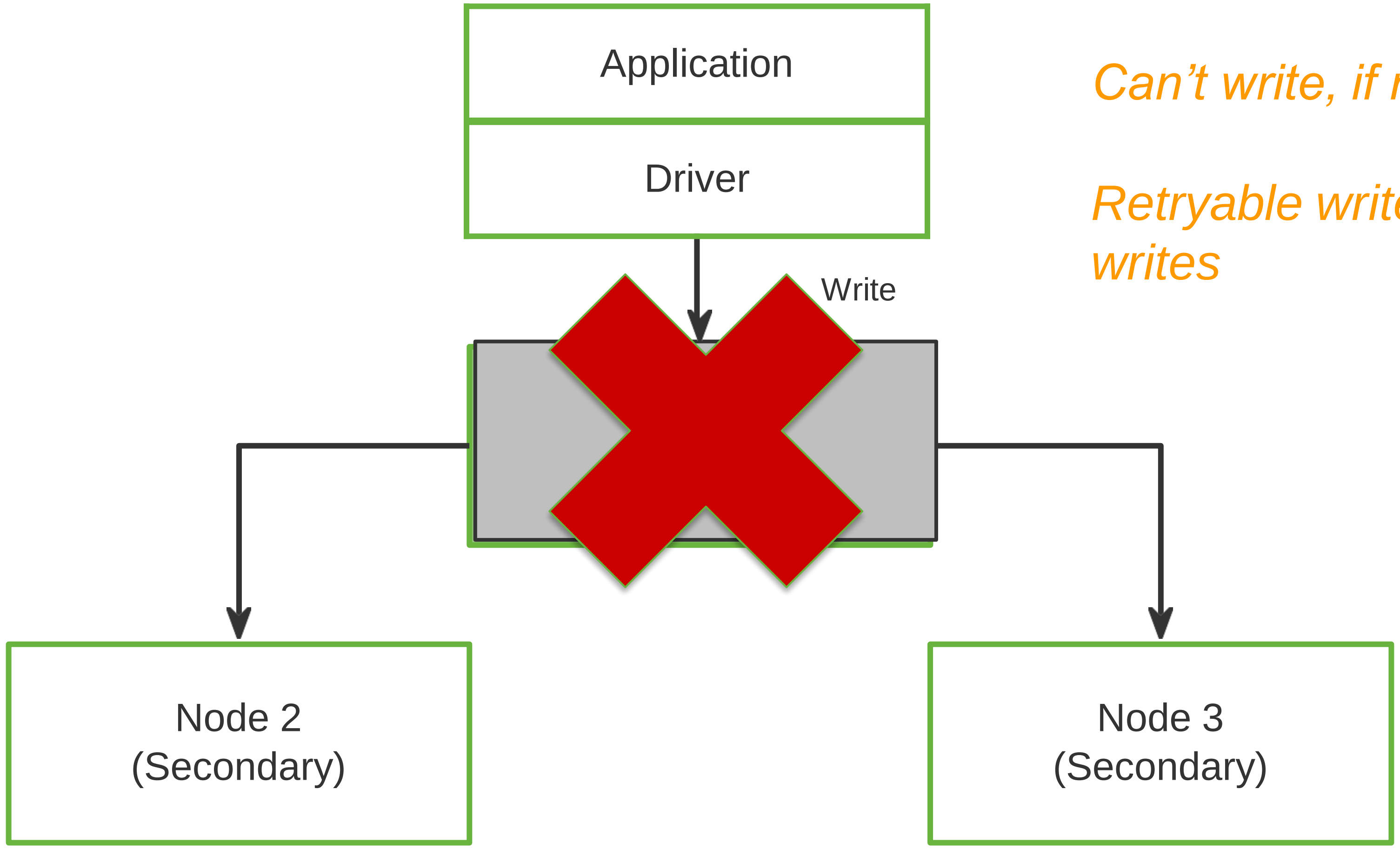
Read from secondary, if no primary

100% read availability

SecondaryPreferred also



Write Availability → Retryable Writes



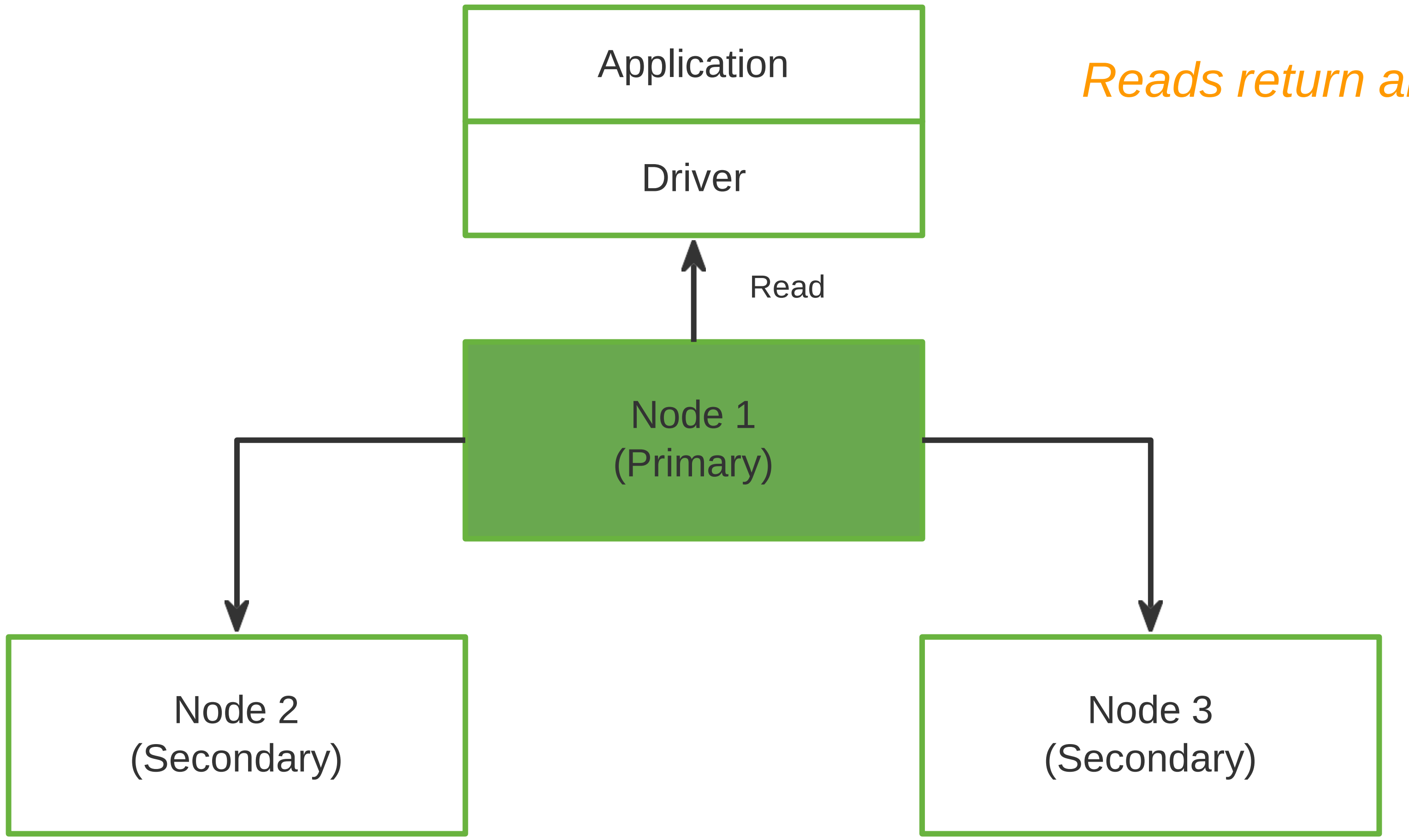
Can't write, if no primary

Retryable writes - retries failed writes

Tuning Consistency

Primary vs. Secondary Reads
Read Concern

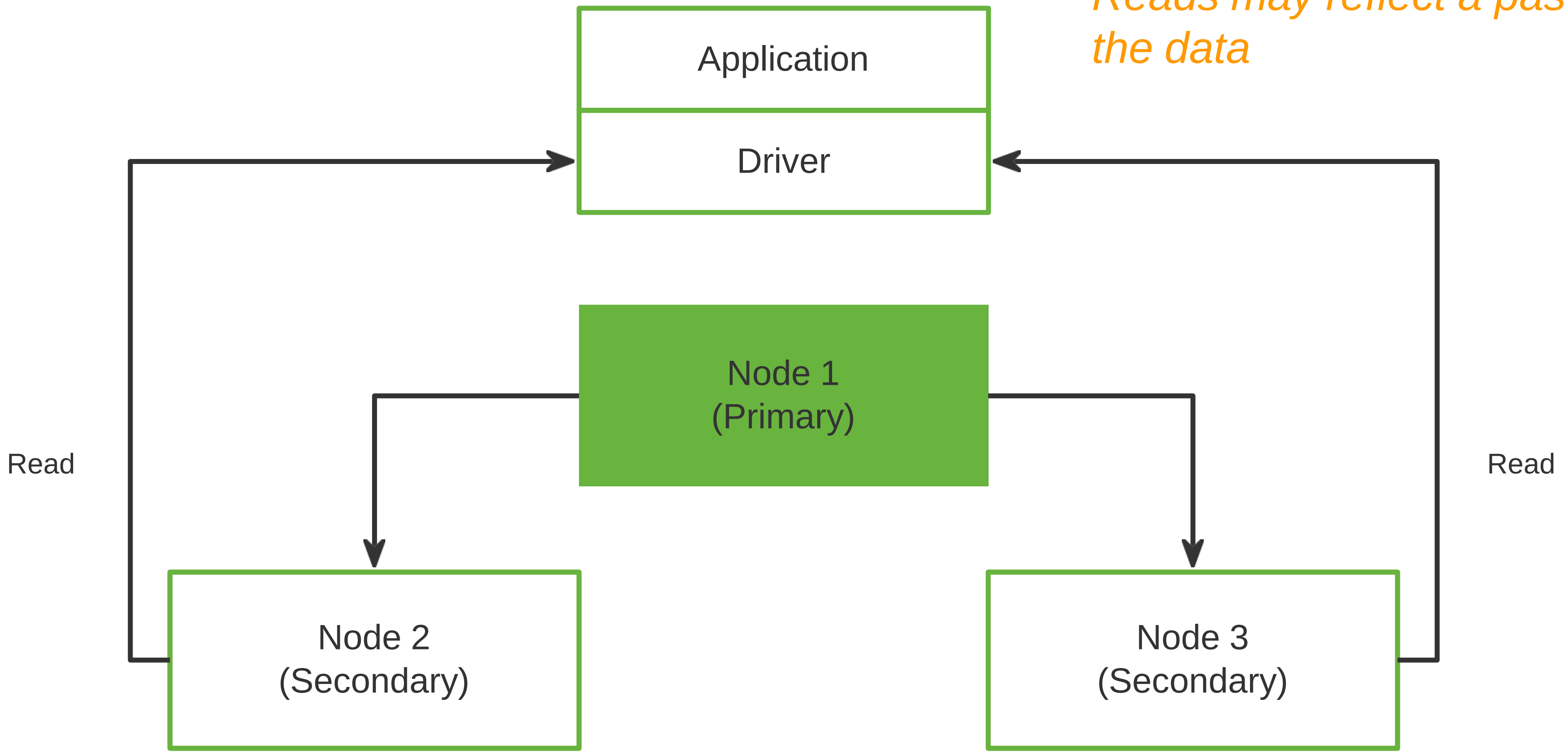
Primary Reads → Strong Consistency



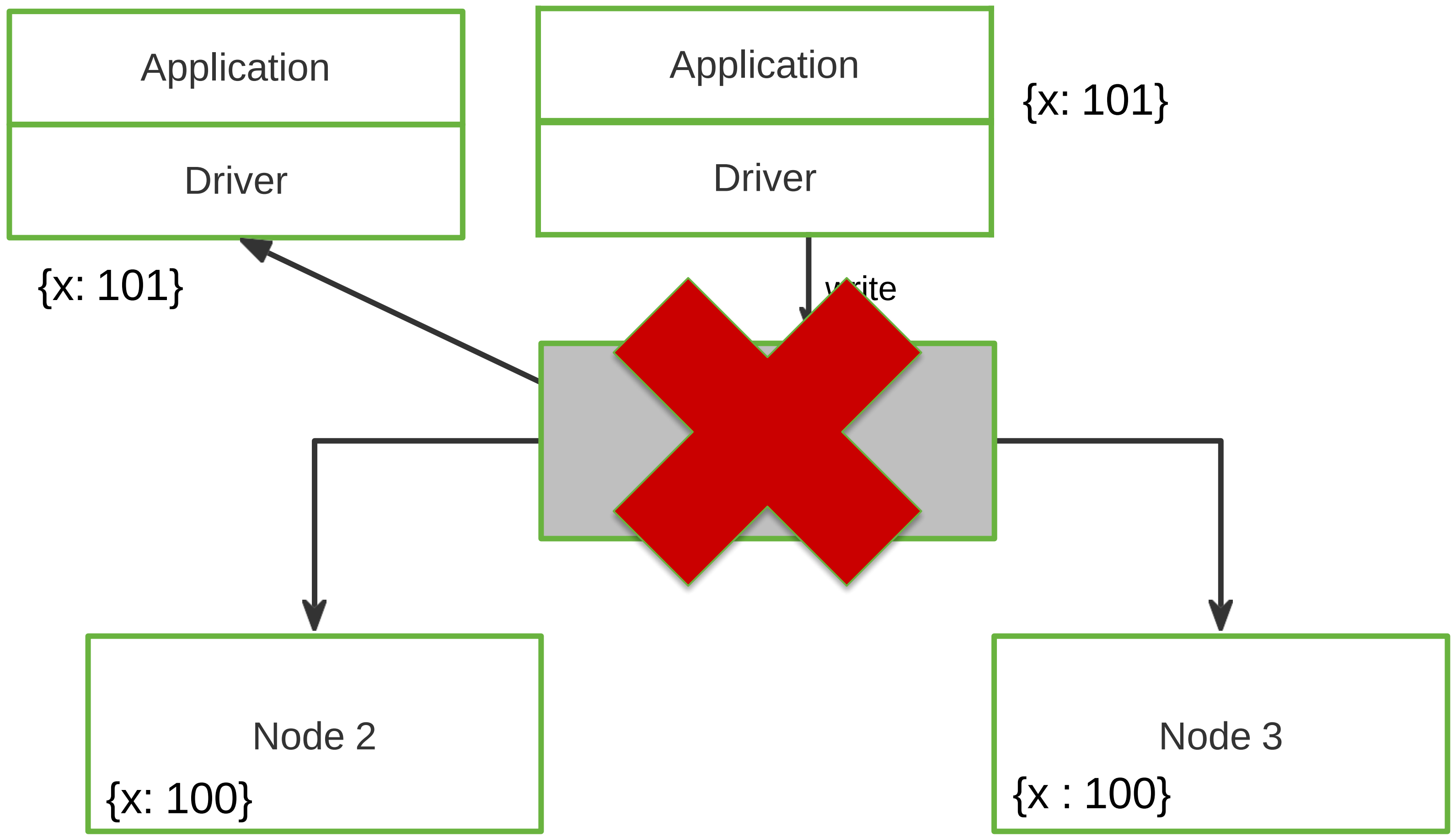
Reads return all previous writes

Secondary Reads → Eventually Consistent

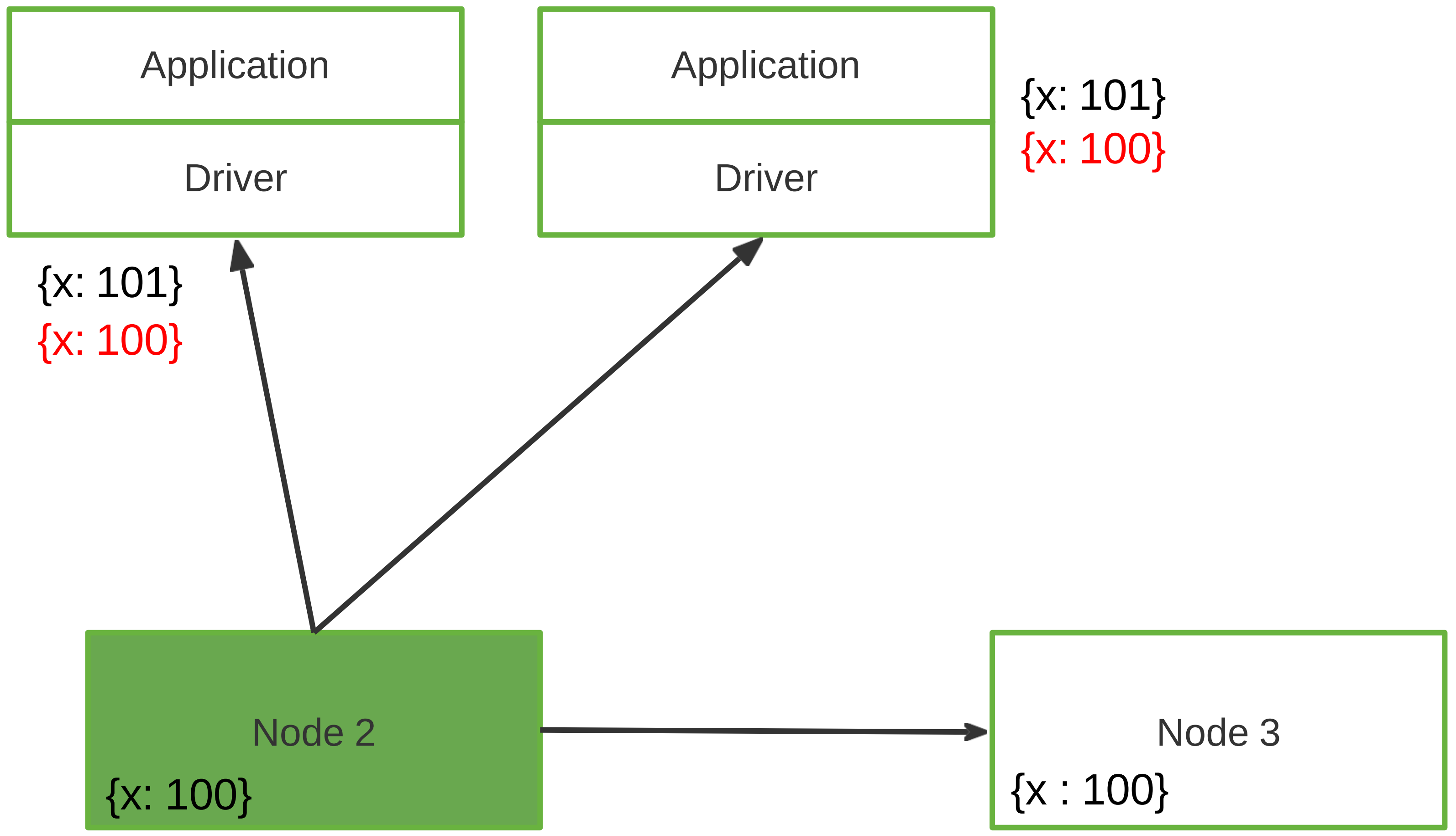
Reads may reflect a past state of the data



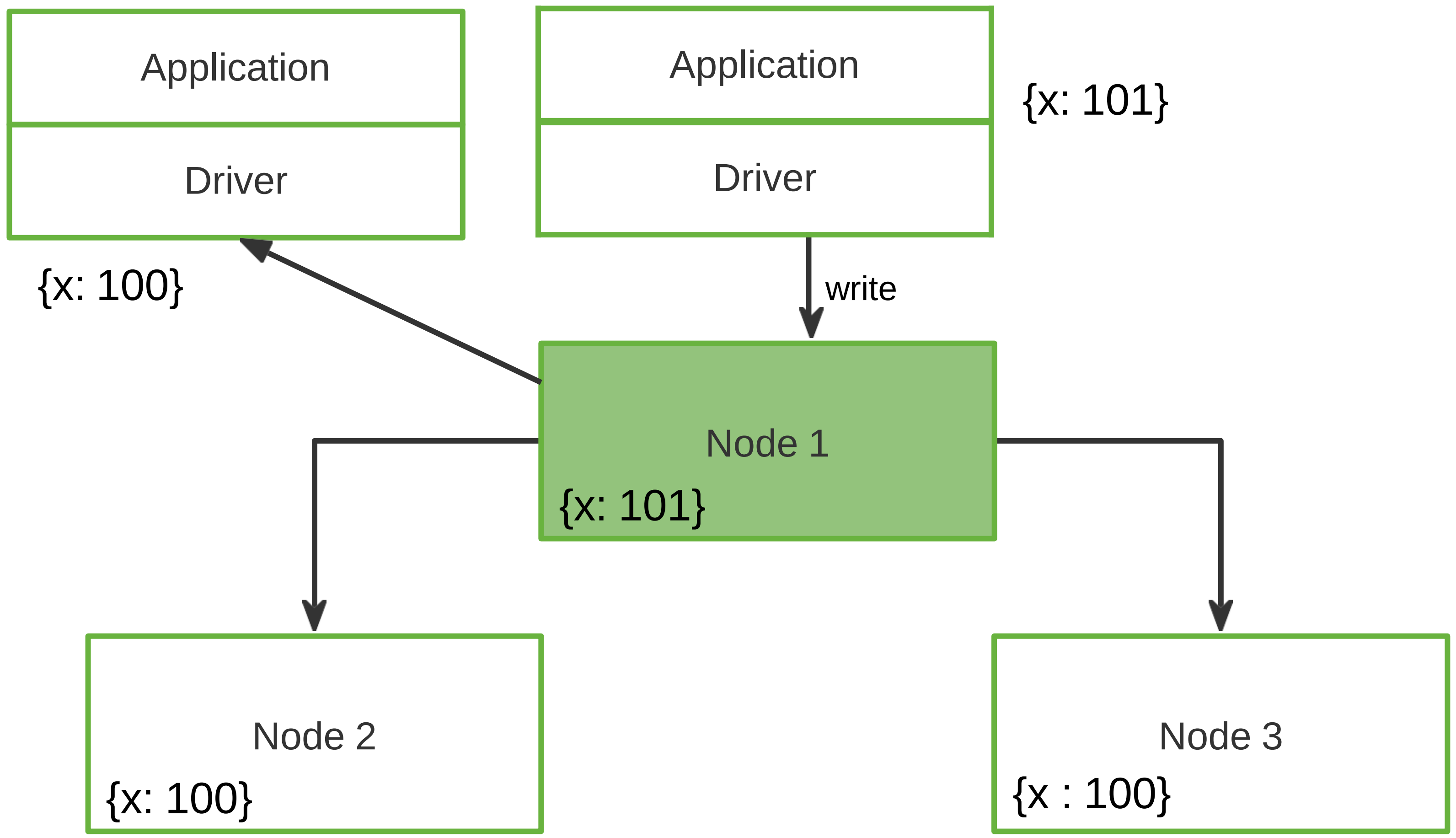
Challenge #1 - Reading Rolled Back Data



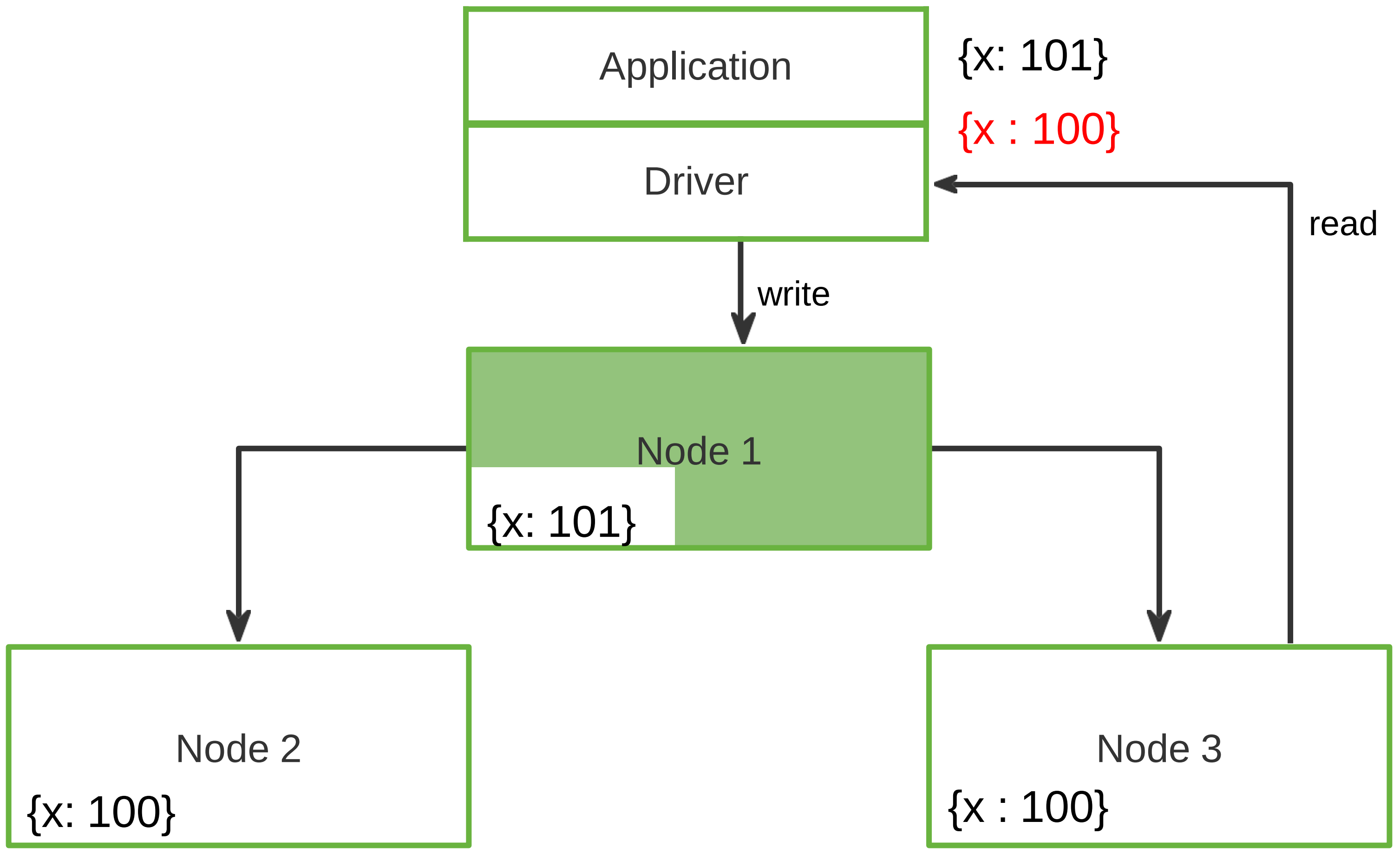
Challenge #1 - Reading Rolled Back Data



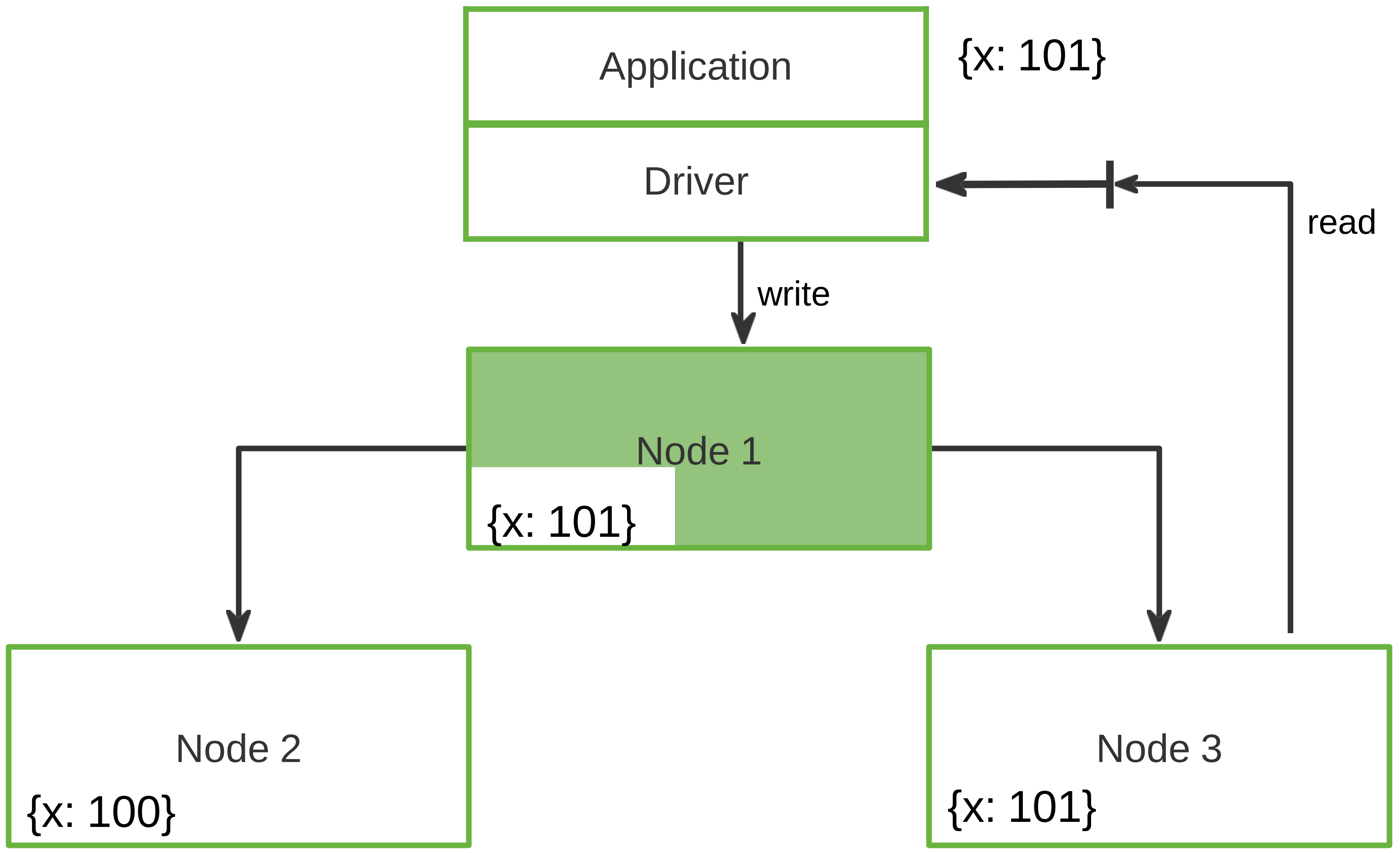
ReadConcern: Majority



Challenge #2 - Secondary Reads Don’t Reflect My Writes



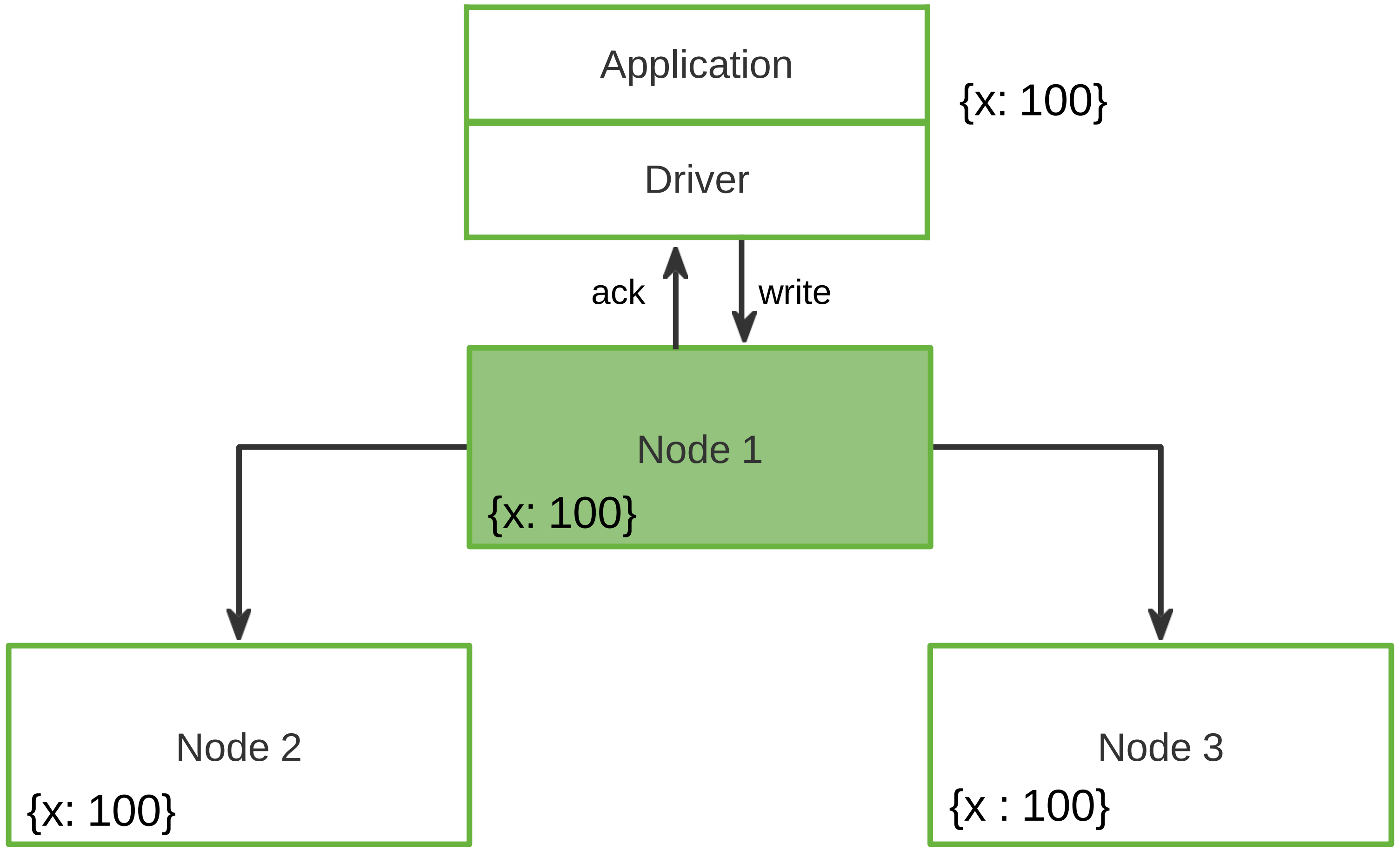
Causal Consistency



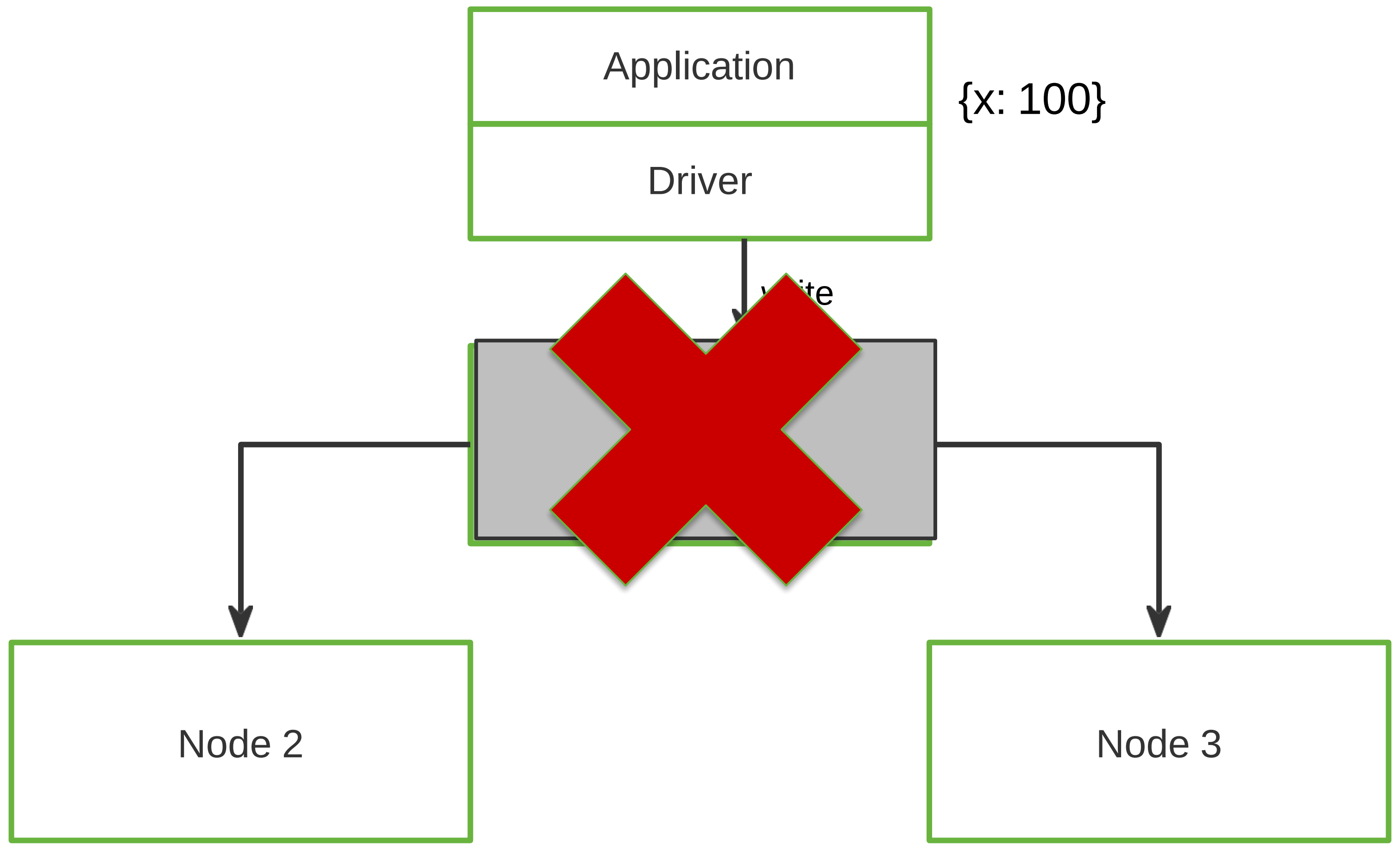
Tuning Durability

Write Concern

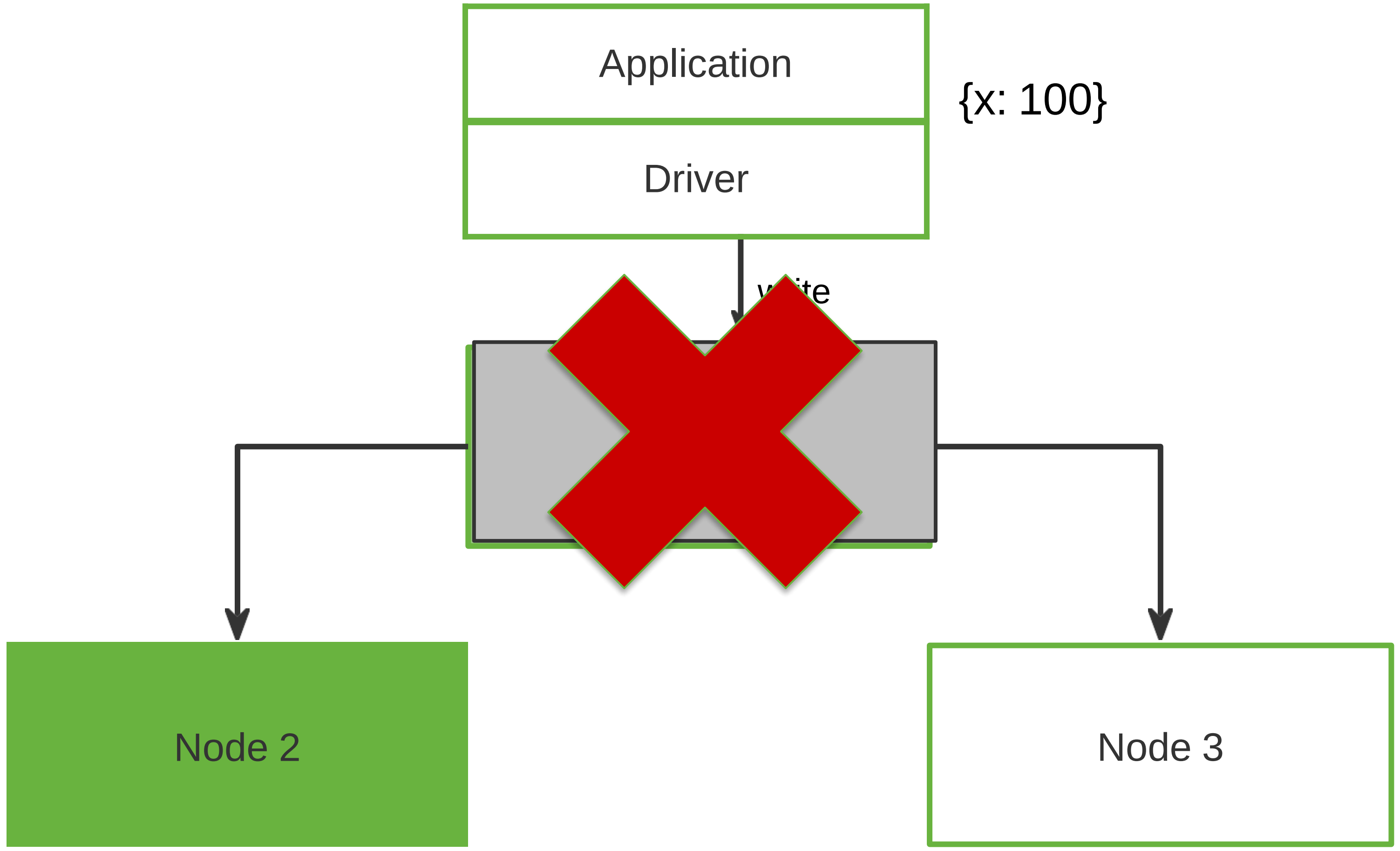
Replica Set Write Durability



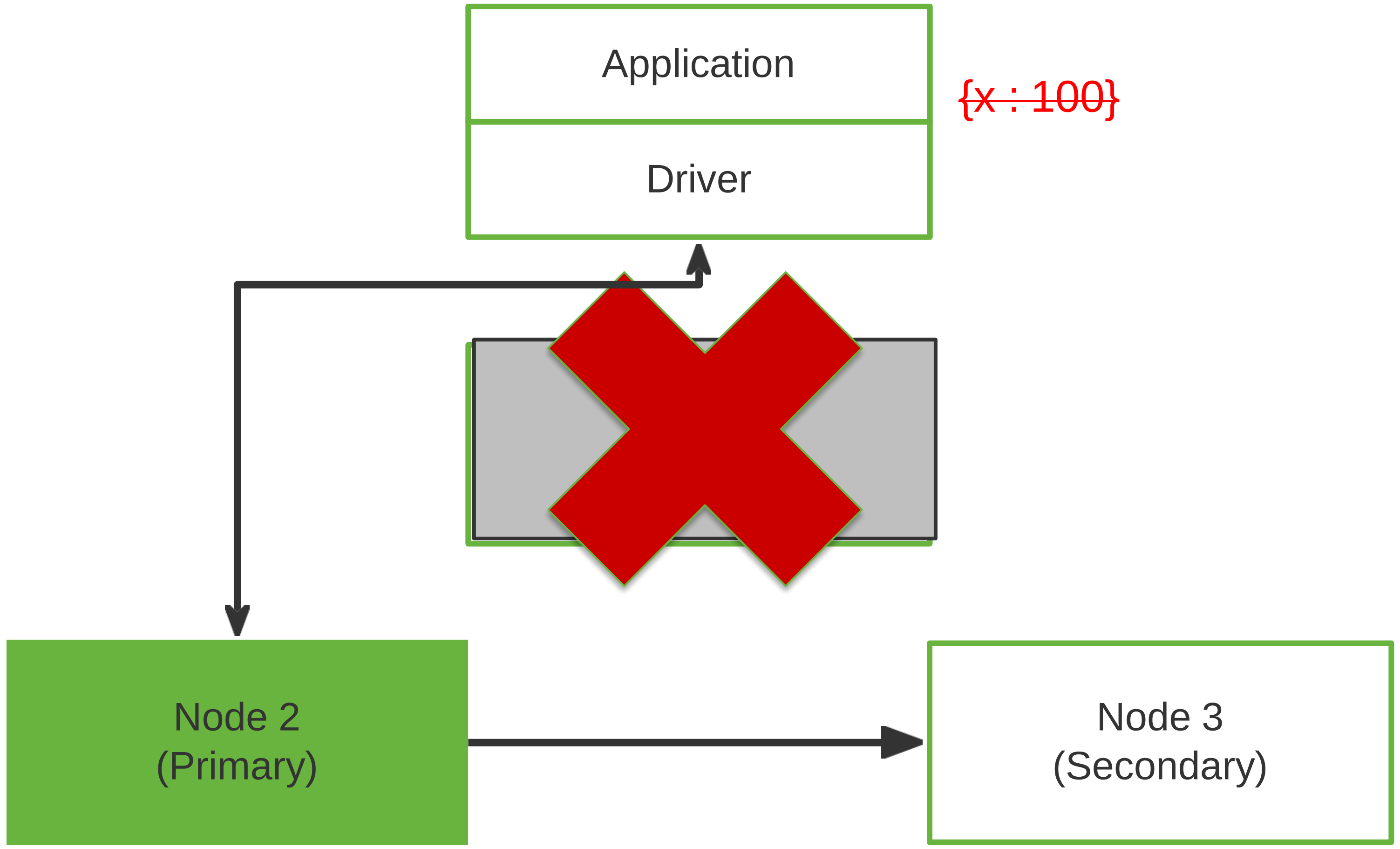
Replica Set Write Durability



Replica Set Write Durability



Write Durability



Write Durability is Configured via Write Concern

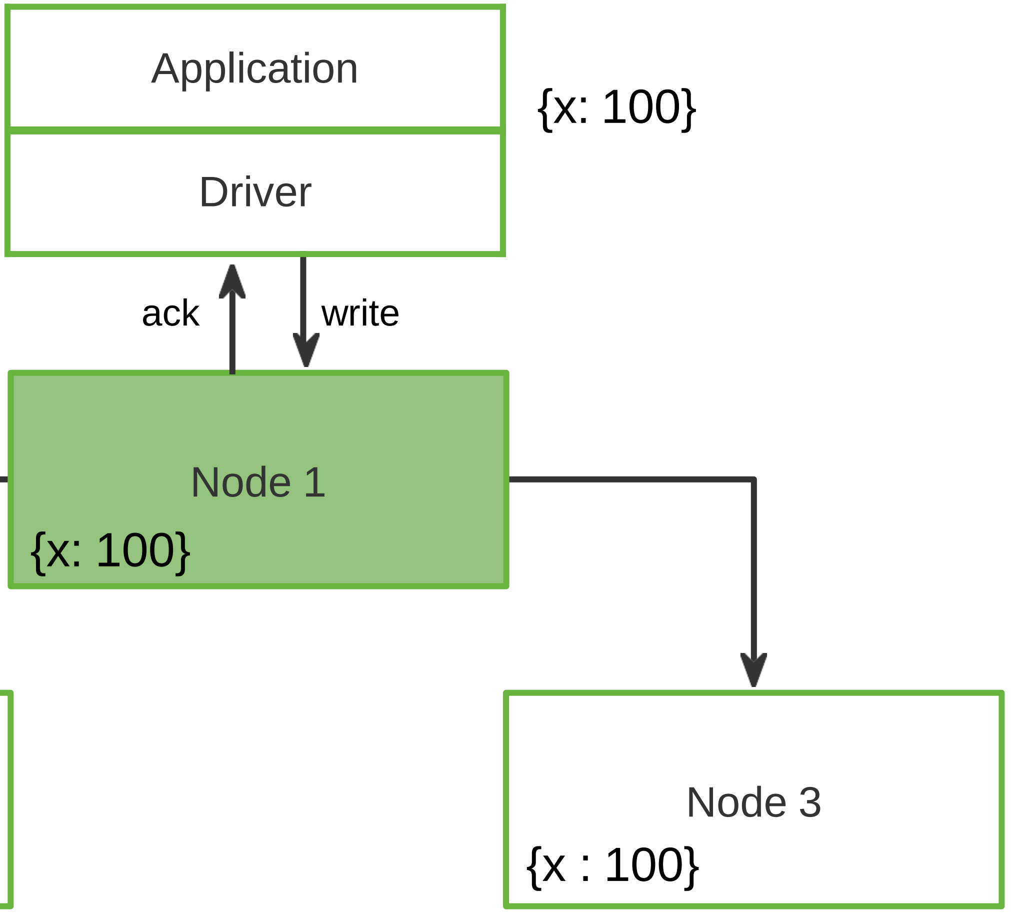
- Intelligent write receipt/confirmation
  - Specifieds the the number of nodes that must have written the write to disk
  - Default number is 1

- NOT a distributed transaction

```
db.test.insert({x:100},{writeConcern:{w:2}})
```

Write Concern Majority

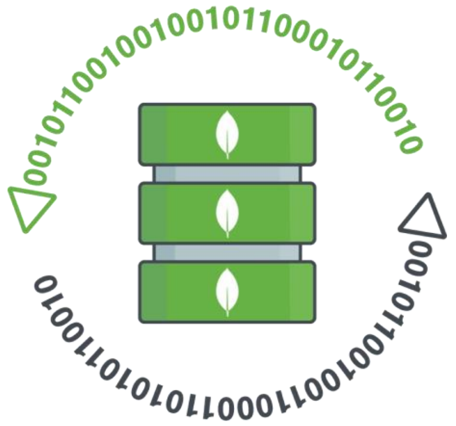
```
db.test.insert({x:100},
{writeConcern:{w:"majority"}})
```



MongoDB Retryable Writes

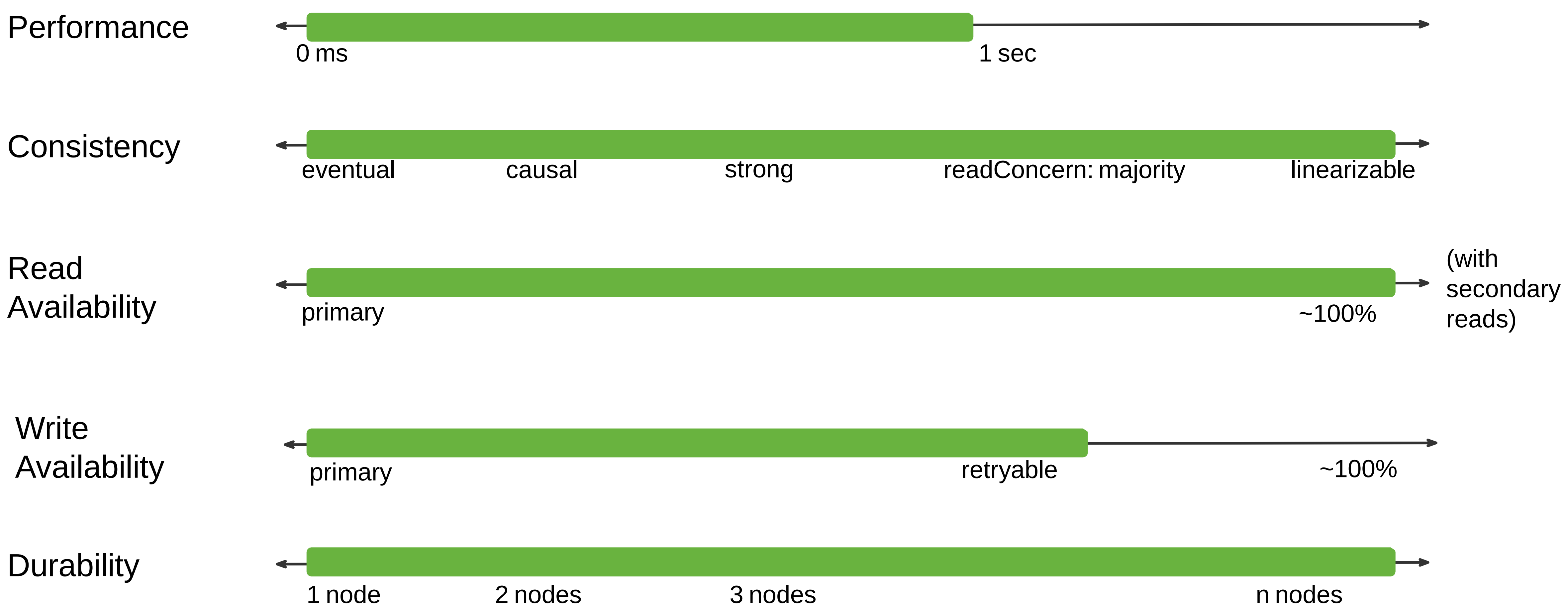
Write failure handling moved from the app to the database for transient network errors or primary elections

- Driver automatically retries failed write
- With a unique transaction identifier, server enforces exactly-once processing semantics

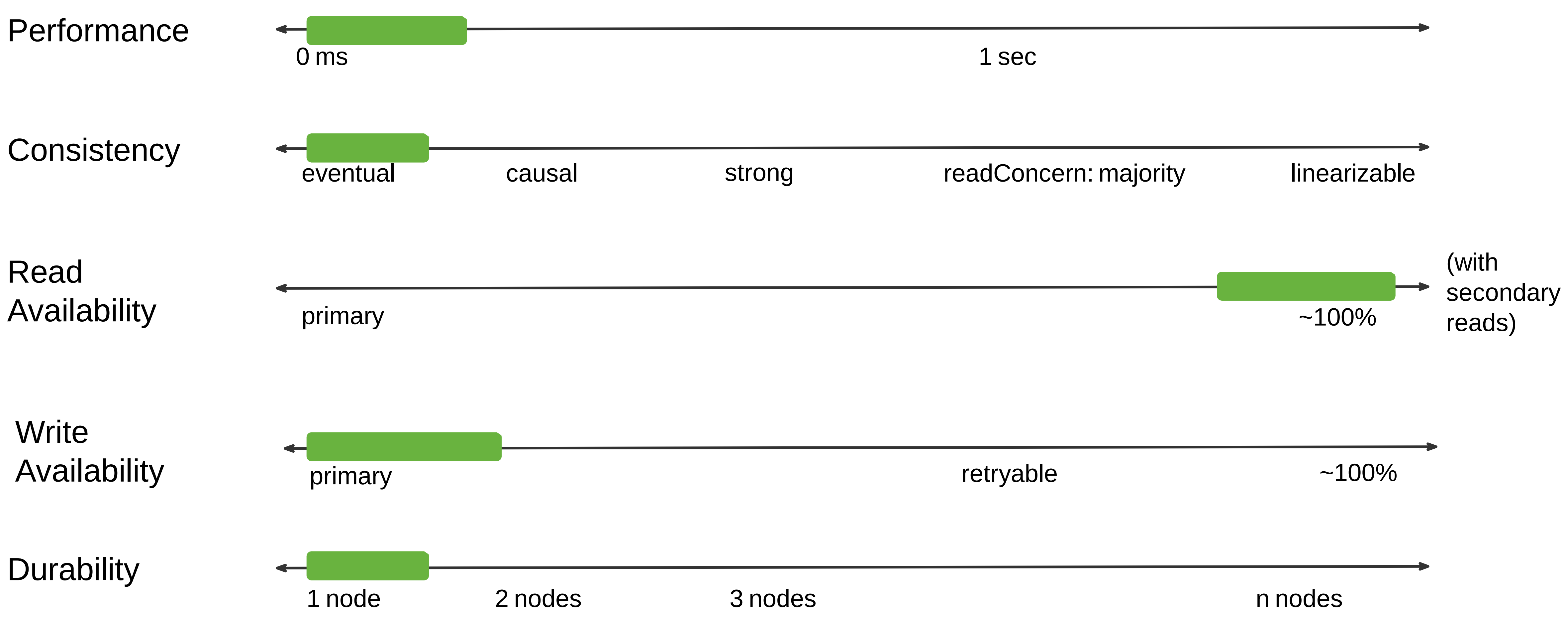


- Properties
  - Supports idempotent & non-idempotent operations, and errors caused by time-outs
- Delivers always-on, global availability of write operations
  - Overcomes the complexity imposed by multi-master, eventually consistent systems

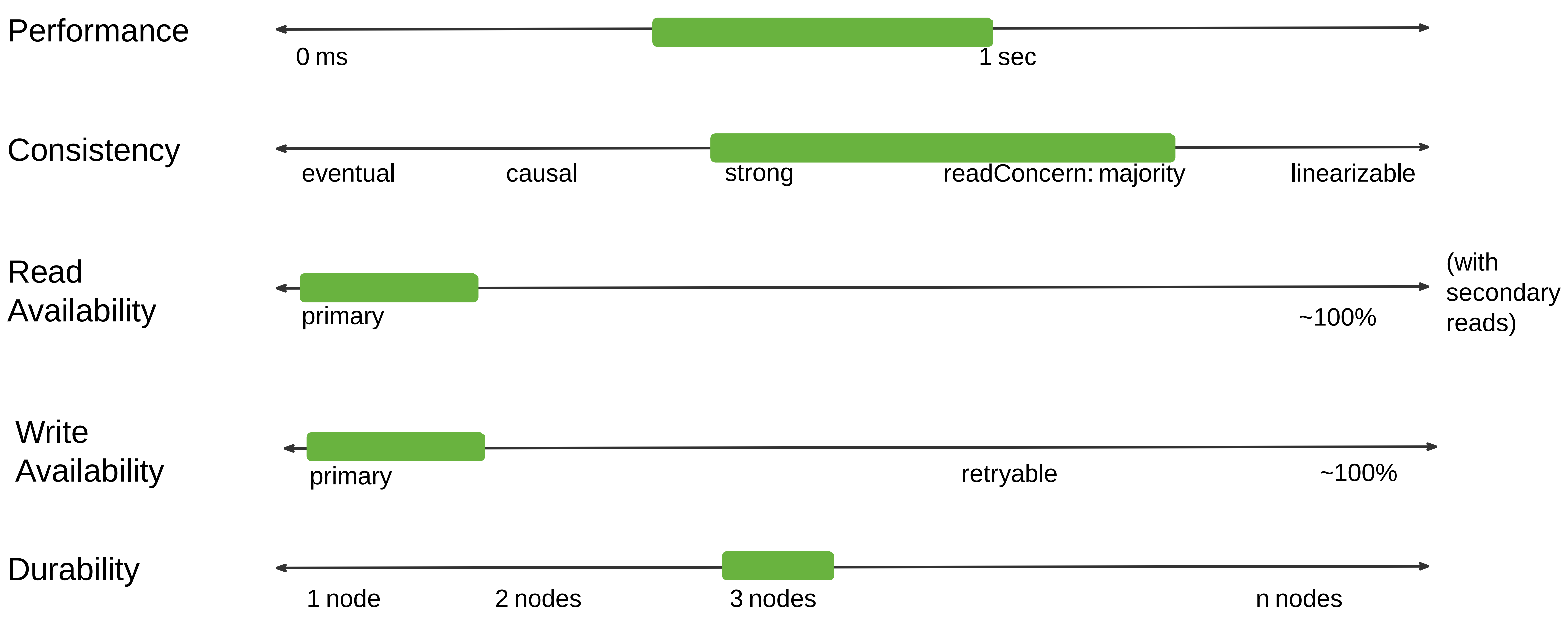
Configuring MongoDB



Performance Optimized



Durability and Consistency Optimized



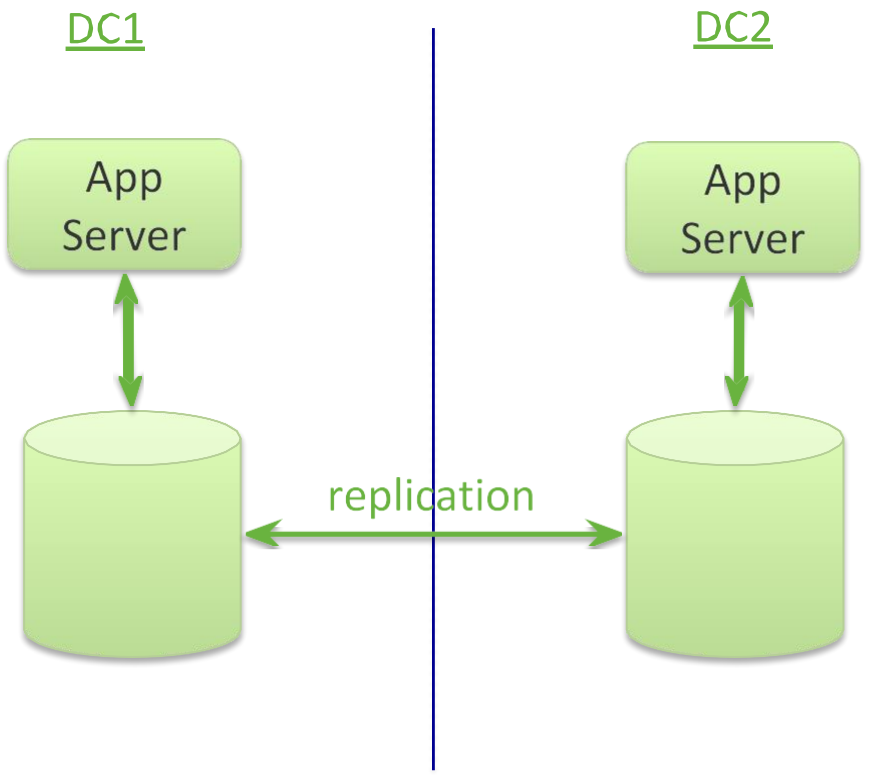
MongoDB Multi-Data Center Deployments

Writes: Multi-Data Center

3 Alternatives:

1. Two-phase commit across data centers
   - Consistent

   •Slow Multi
2. master
   - Each data item can be written to any data center
   - Inconsistent - Conflict resolution and data loss
   - Fast
3. Sharded Database
   - Multiple masters; each document/partition has a single master
   - Multiple masters can be deployed in various data centers
   - Consistent
   - Fast



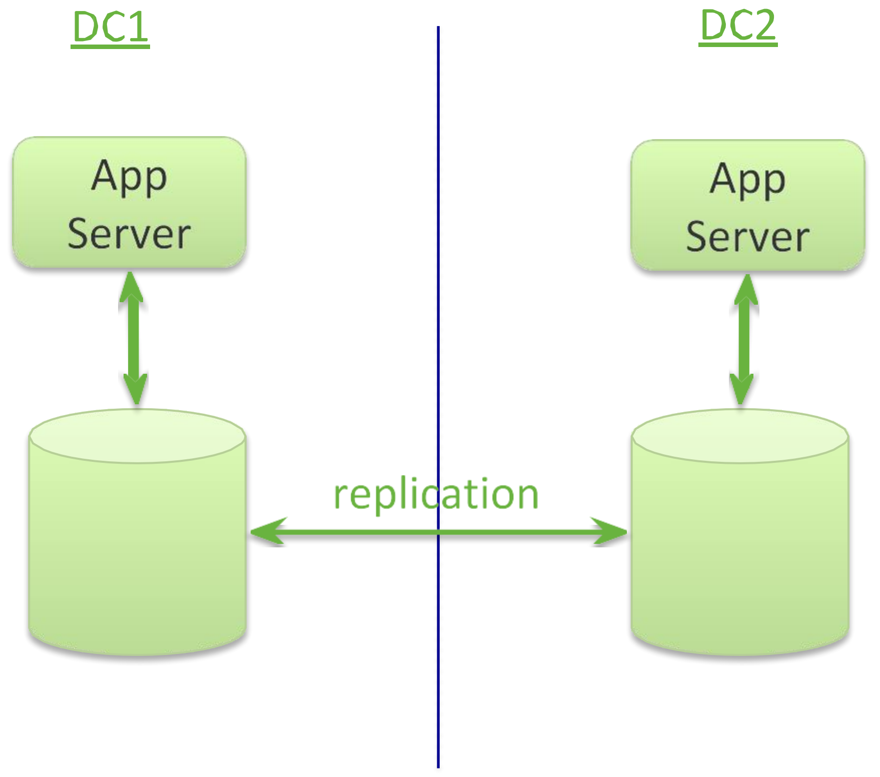
Writes: Multi-Data Center

3 Alternatives:

1. Two-phase commit across data centers
   - Consistent
   - Slow
2. Multi master
   - Each data item can be written to any data center
   - Inconsistent - Conflict resolution and data loss
   - Fast
3. Sharded Database
   - Multiple masters; each document/partition has a single master
   - Multiple masters can be deployed in various data centers
   - Consistent
   - Fast

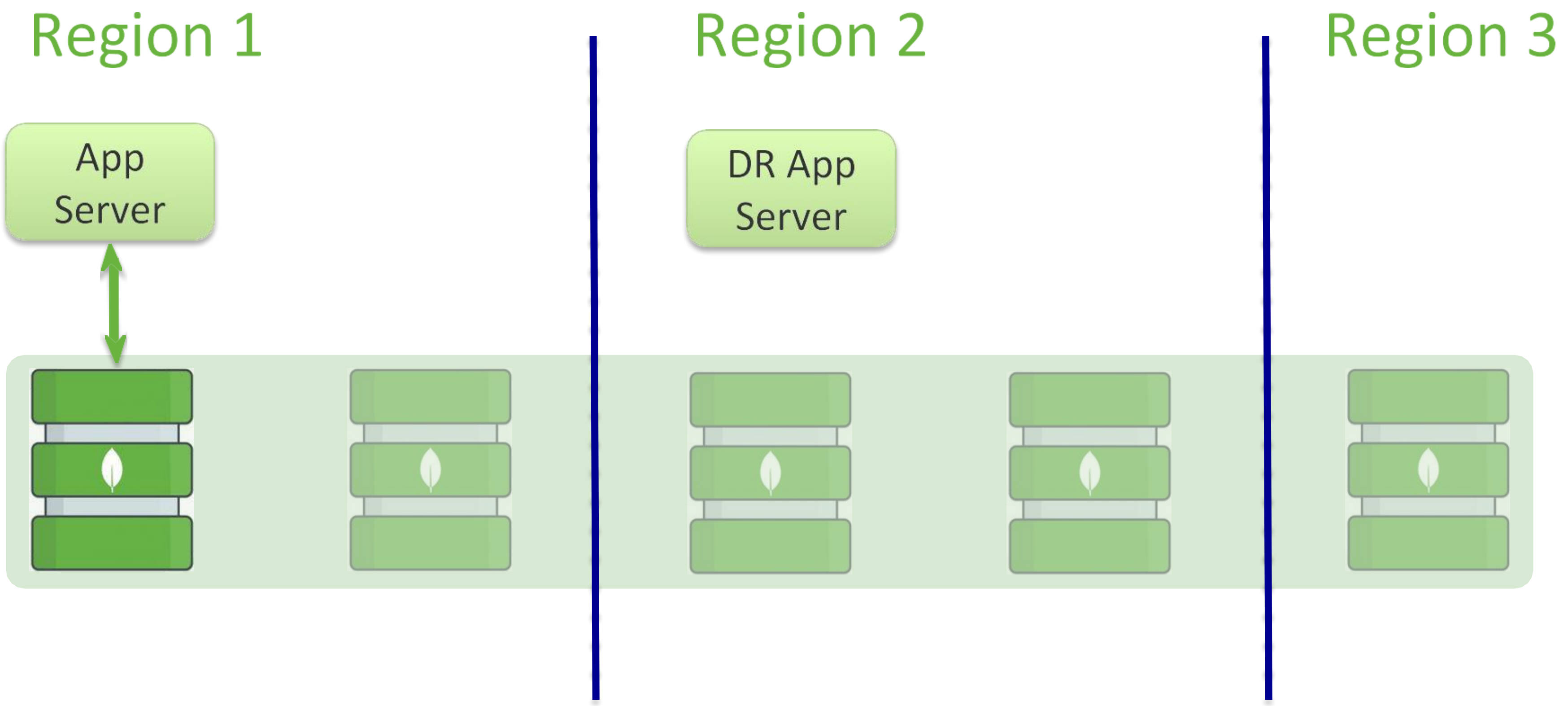
MongoDB???

MongoDB

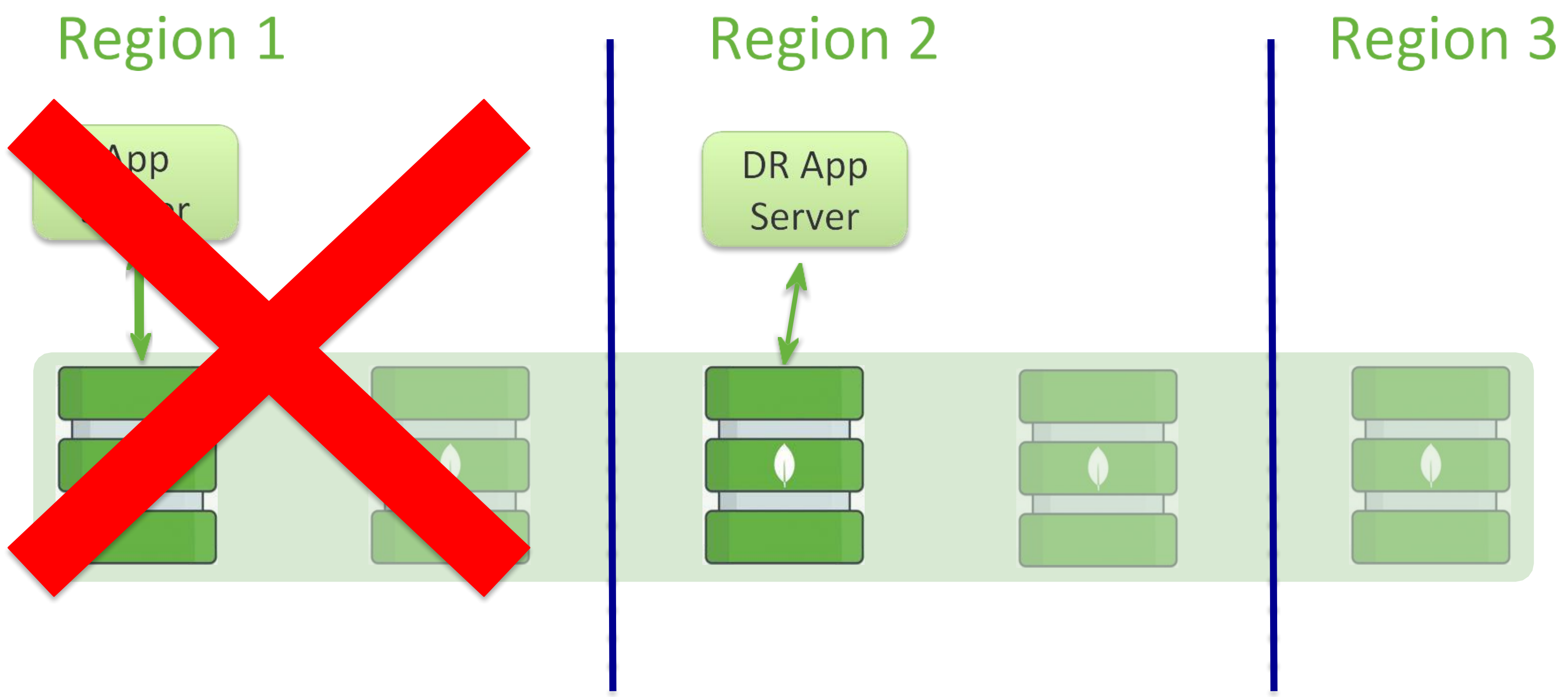


Option 1: Active DR

Multi Region



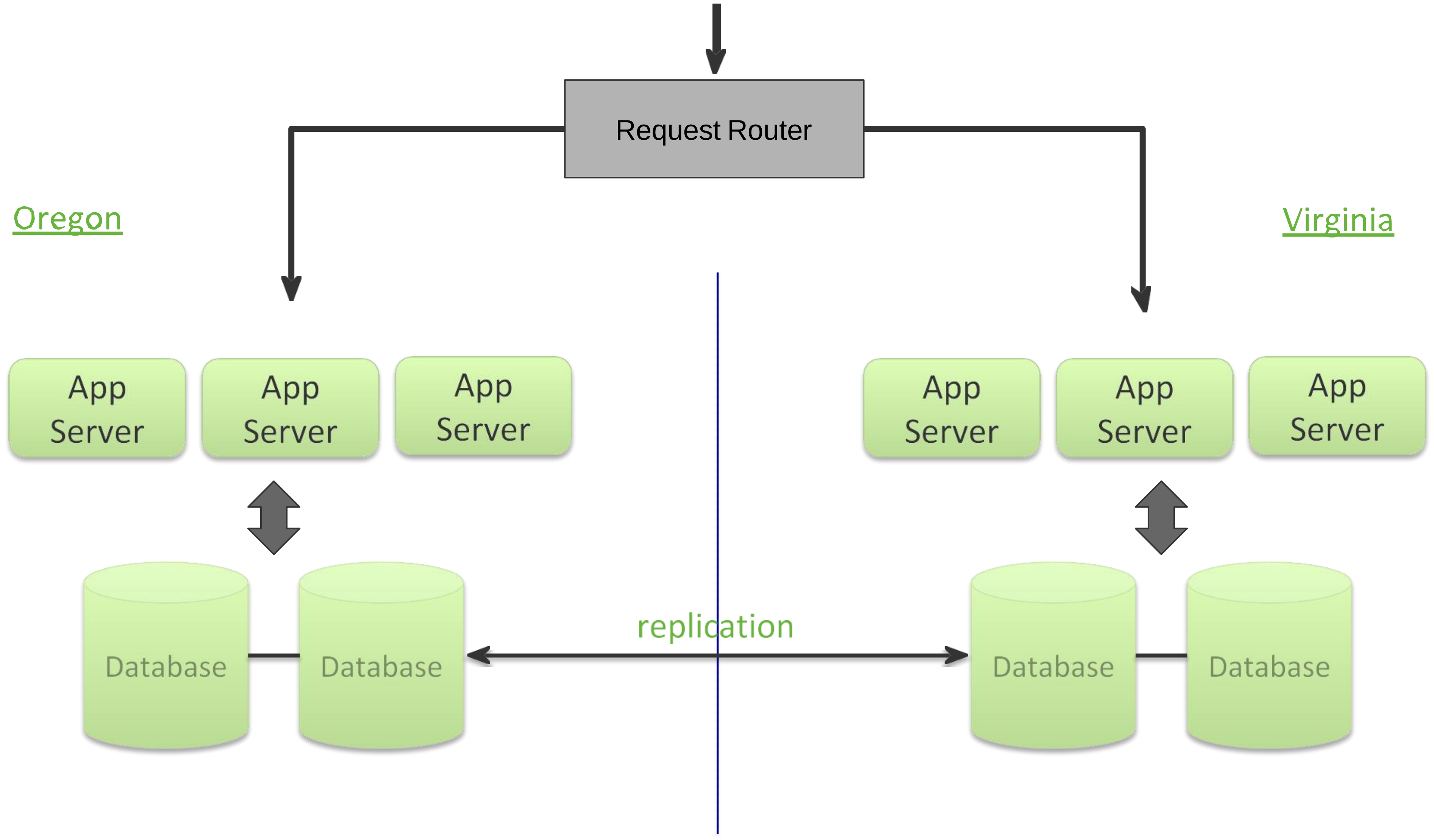
Multi Region



Tune ReadPreference, ReadConcern, WriteConcern, Causal Consistency

Option 2: Sharded Database

Routing Requests To Nearest Region



Sharded Database (zone sharding)

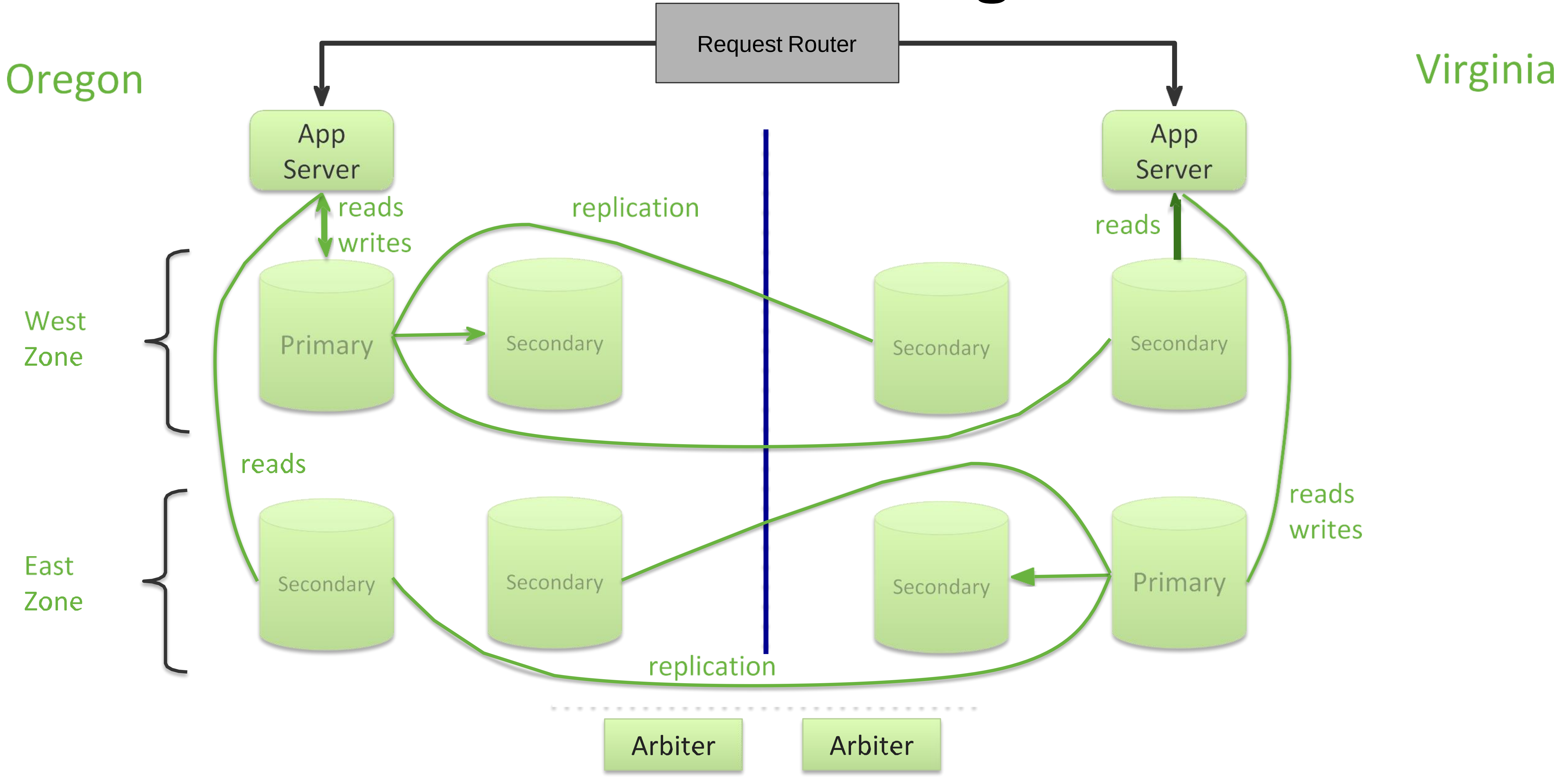
Writes

- Each data center owns a partition of the data
- External routing process routes requests to the data center owning the request's data
- Application writes to local primary
- Occasional cross data writes may occur if routing process isn't perfect

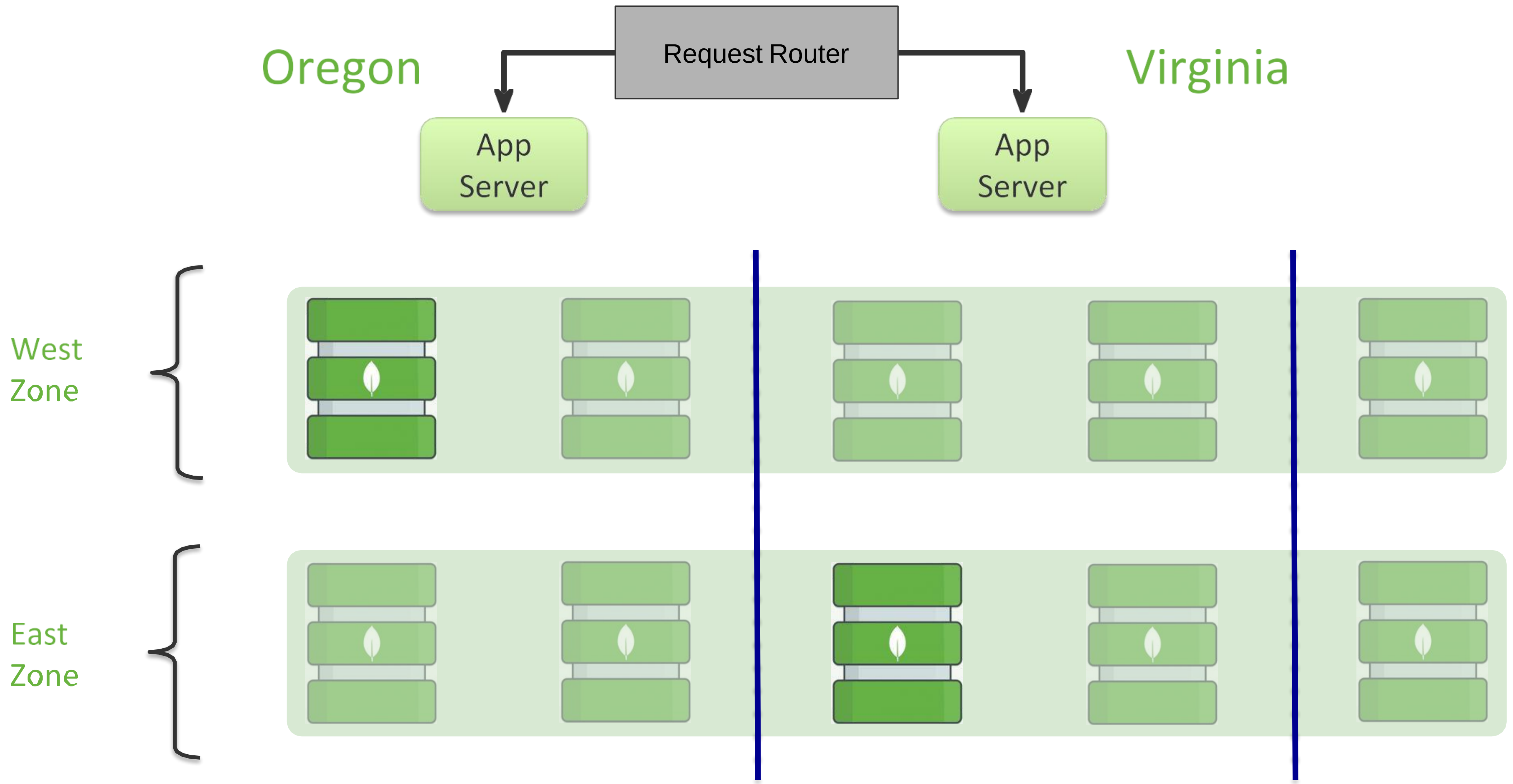
Reads

- Read from local primary for consistency
- Global data access using "nearest" read preference. Eventually consistent.

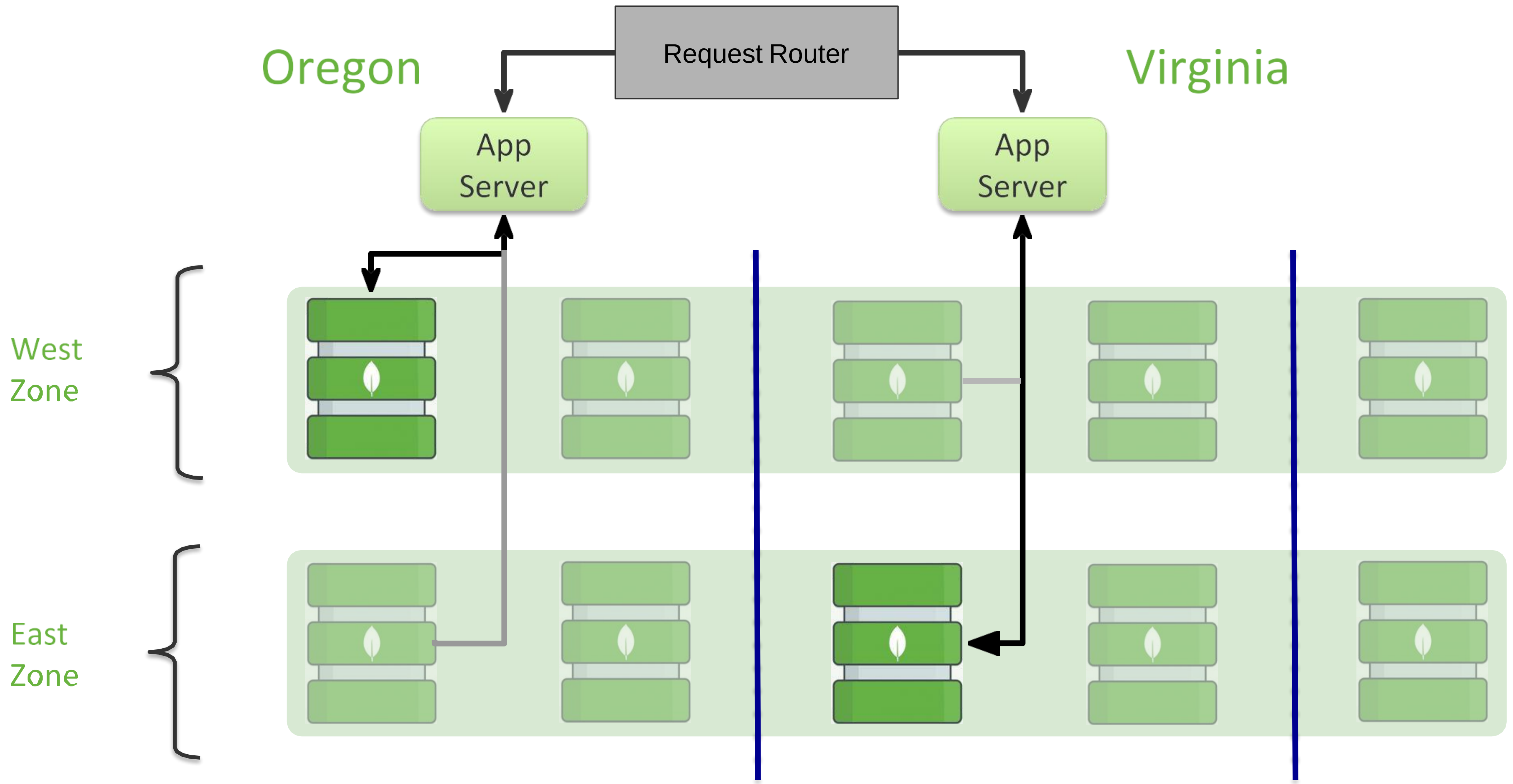
Zone Sharding



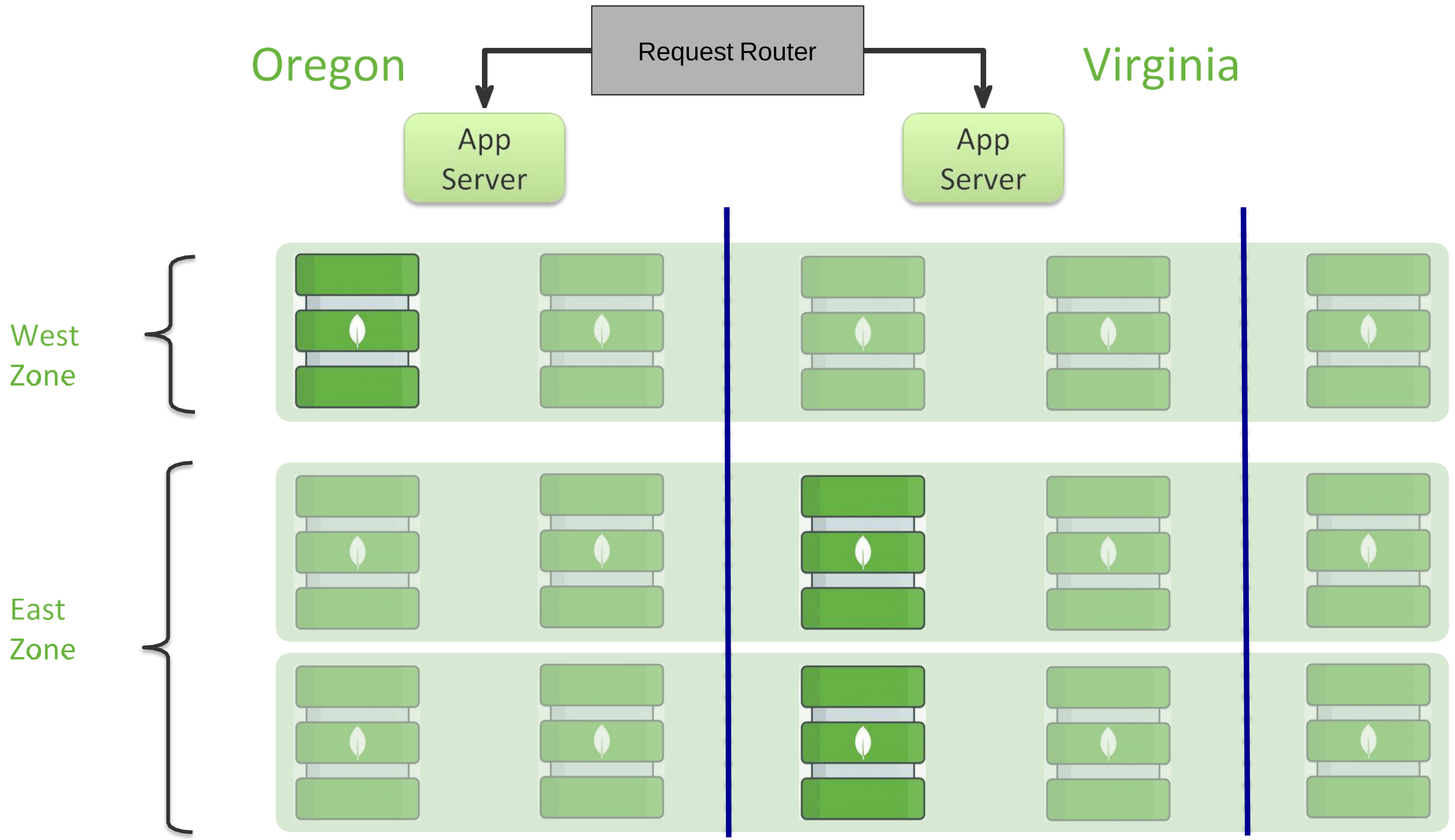
Zone Sharding



Zone Sharding



Zone Sharding

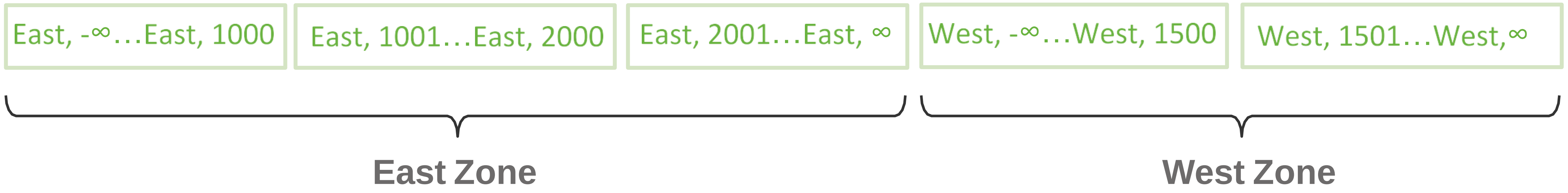


Location-Aware Data Distribution

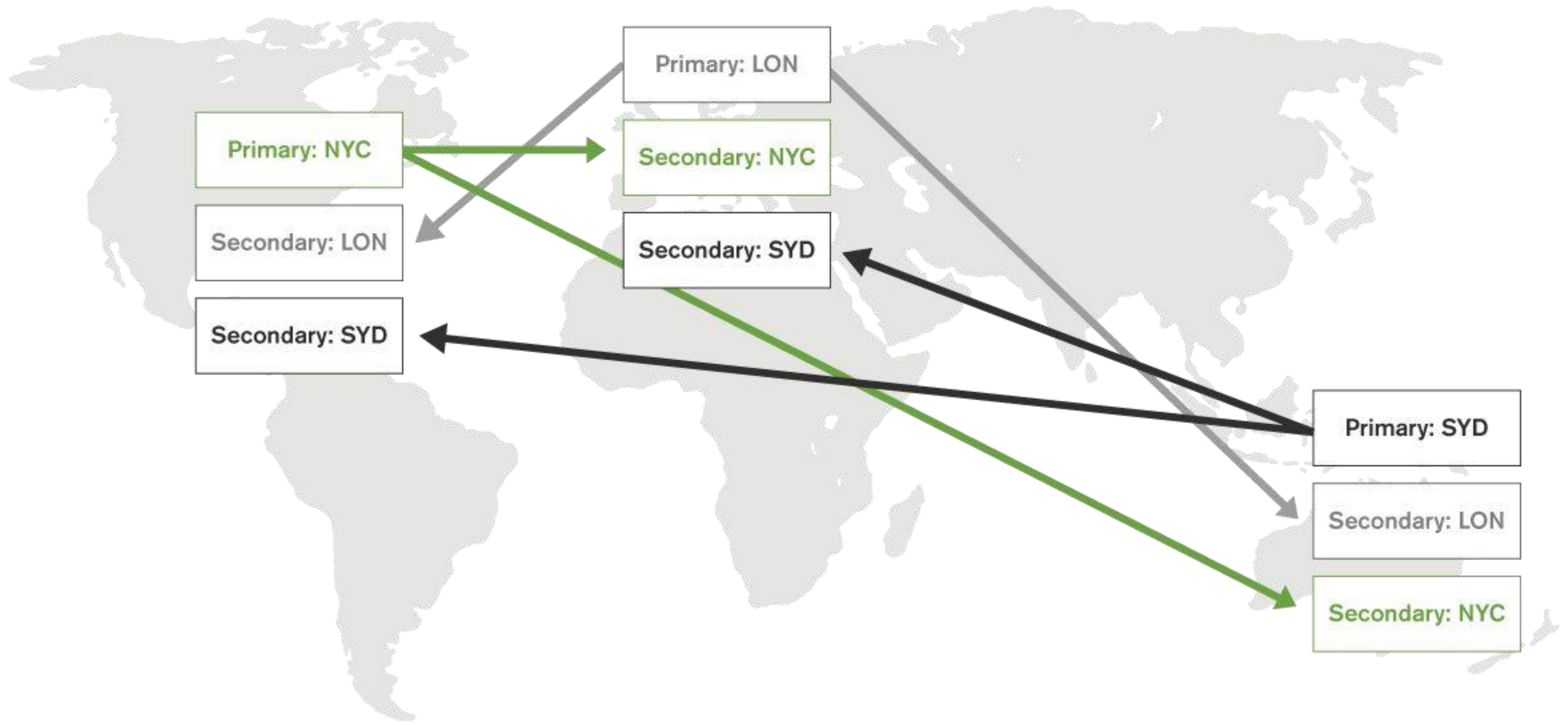
Shard Key: **{regionCode} {userId}**

Zone	Start	End
West	West, MinKey	West, MaxKey
East	East, MinKey	East, MaxKey

```
{
   _id : ObjectID("abc…"),
   regionCode: "West",
   userId: 12345,
   …
}
```



Read Global/Write Local



Option 3: Multi-Master Database

How to implement Multi-Master

1. Insert only
   - Updates → Inserts
   - Aggregate on read
2. Always write to local primary
3. All reads are scatter gather

Updates

```
db.carts.find({shopCartId: 1234})



db.carts.update(
   {shopCartId: 1234},
   {$push: {items: “socks12”}})
```

ShoppingCart Collection

```
{
  shopCartId: 1234,
  items: [“shoe32”,
          “coat43”]
}
```

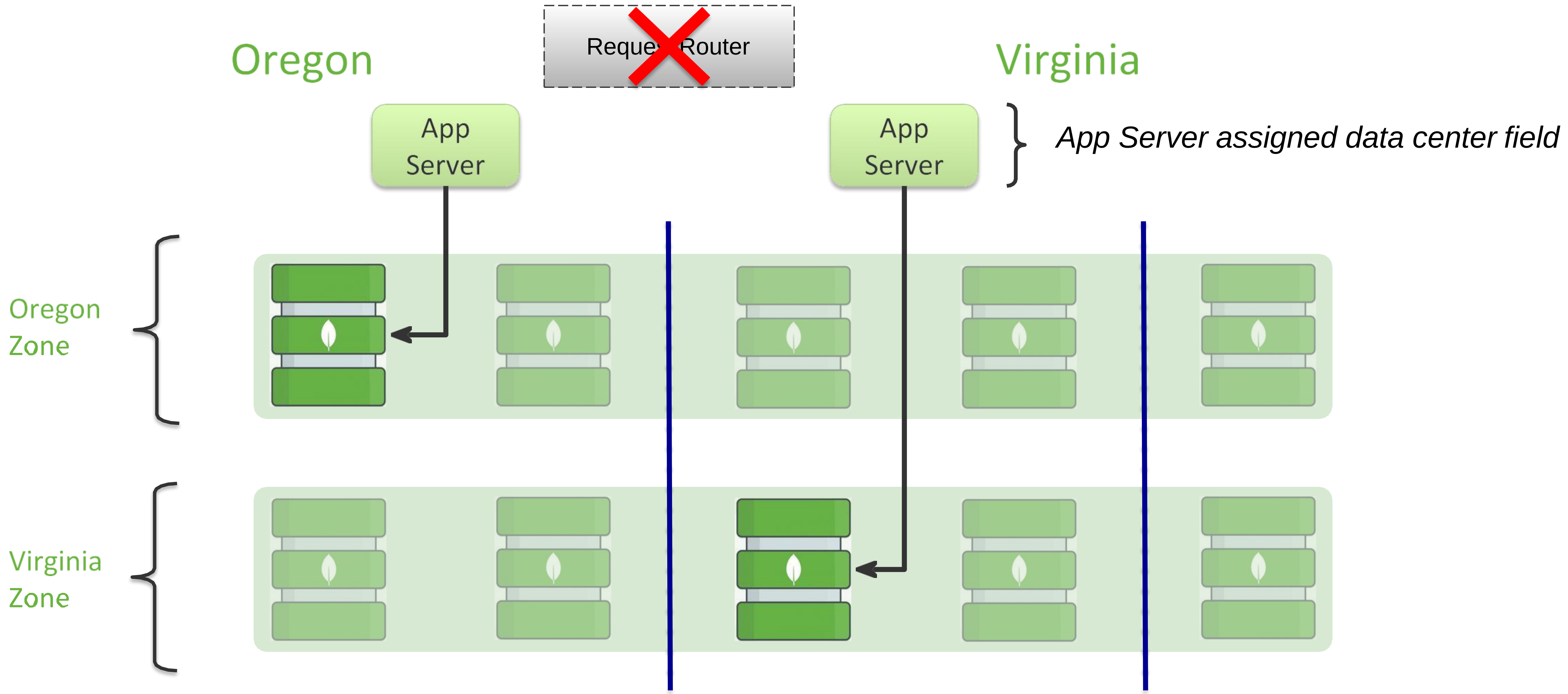
Insert only

```
db.carts.aggregate(
  [{$match: {shopCartId: 1234,
             op: “add”}},
   {$group: {_id: “$shopCardId”,
             items: {$push:
                     “$items”}}}]


db.carts.insert(
   {shopCartId: 1234,
    op: “add”,
    items: “socks12”}
)
```

CartOperations Collection

```
{
  shopCartId: 1234,
  op: “add”,
  items: “shoe32”
}

{
  shopCartId: 1234,
  op: “add”,
  items: “coat32”
}
```

Multi-Master Writes (inserts)

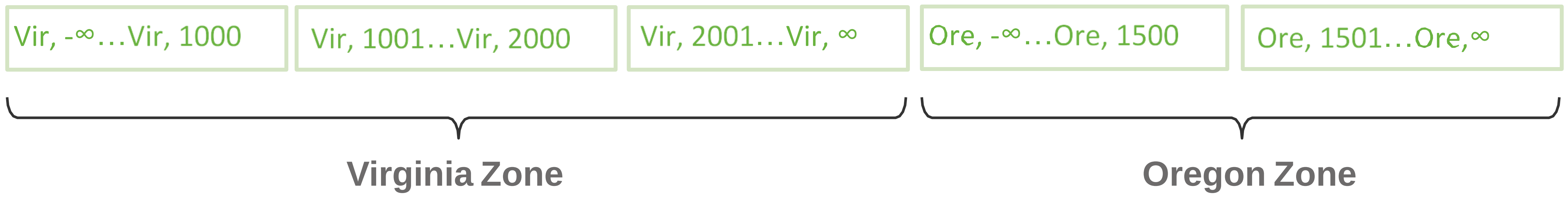


Mult-master zone sharding configuration

Shard Key: **{dc} {shopCartId}**

Zone	Start	End
Virginia	Virginia, MinKey	Virginia, MaxKey
Oregon	Oregon, MinKey	Oregon, MaxKey

```
{
  shopCartId: 1234,
  dc: “Virginia”,
  op: “add”,
  items: “shoe32”
}
```



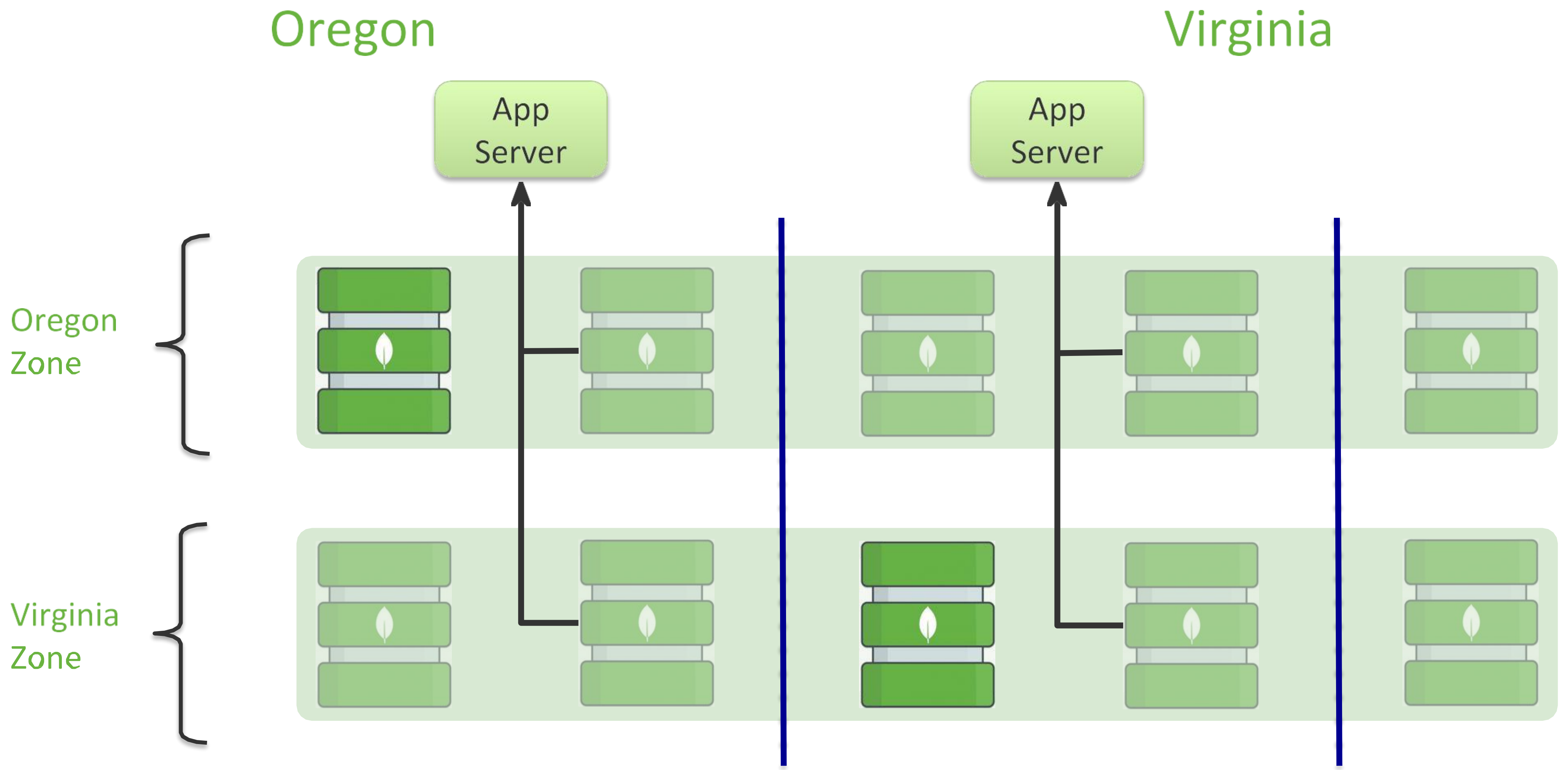
Insert only

```
db.carts.aggregate(
   [{$match: {shopCartId: 1234,
                op: "add"}},
    {$group: {_id: "$shopCartId",
               items: {$push : "$items"}}}]


db.carts.insert(
    {shopCartId: 1234,
     dc: "Oregon",
     op: "add",
     items : "socks12"}
)
```

CartOperations Collection

```
{
  shopCartId: 1234,
  dc: "Oregon",
  op: "add",
  items: "shoe32"
}

{
  shopCartId: 1234,
  dc: "Virginia"
  op: "add",
  items: "coat32"
}
```

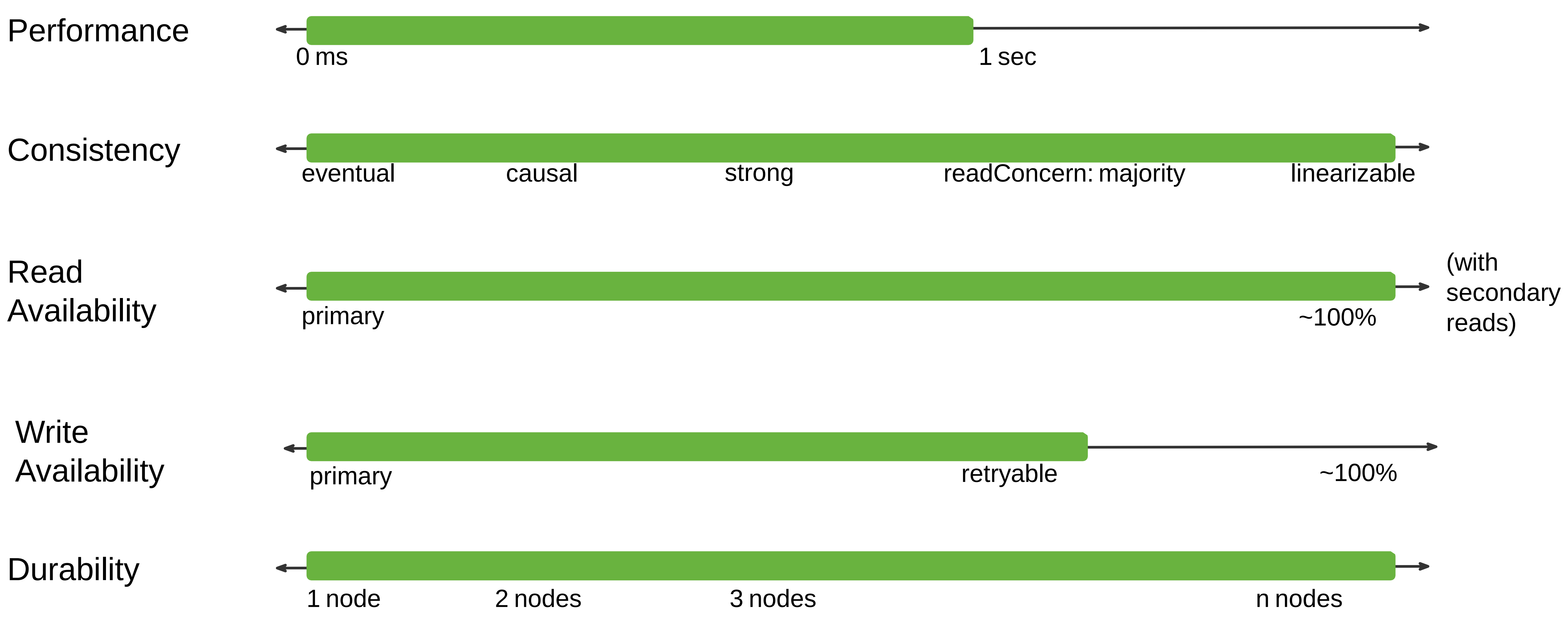
Multi-Master Reads (scatter gather)



Multi-Data Center Approach Summary

Architecture	Advantages	Disadvantages
Active - DR	Simple to implement Consistent	Long latency writes for some users
Sharded Database	Active Active Consistent No conflict resolution/data loss	Requires external routing process Need to configure zone sharding
Multi-Master	Active Active No external routing process • Easier for architects	Eventually consistent • More complexity for developers May not work for all use cases Slower/more complex reads • Scatter gather aggregates

Tune to application requirements


Performance
0 ms
1 sec
Consistency
eventual
causal
strong
readConcern: majority
linearizable
Read Availability
primary
~100%
(with secondary reads)
Write Availability
primary
retryable
~100%
Durability
1 node
2 nodes
3 nodes
n nodes